
GLPI Developer Documentation Documentation

Teclib'

Mar 27, 2024

CONTENTS

| | | |
|----------|---|-----------|
| 1 | Source Code management | 3 |
| 1.1 | Versioning | 3 |
| 1.2 | Backward compatibility | 3 |
| 1.3 | Branches | 3 |
| 1.4 | Testing | 4 |
| 1.5 | File Hierarchy System | 4 |
| 1.6 | Workflow | 6 |
| 1.7 | Unit testing (and functional testing) | 8 |
| 2 | Coding standards | 11 |
| 2.1 | Call static methods | 11 |
| 2.2 | Static or Non static? | 11 |
| 2.3 | Comments | 12 |
| 2.4 | Variables types | 14 |
| 2.5 | Quotes / double quotes | 15 |
| 2.6 | Checking standards | 16 |
| 3 | Developer API | 17 |
| 3.1 | Main framework objects | 17 |
| 3.2 | Database | 21 |
| 3.3 | Search Engine | 36 |
| 3.4 | Massive Actions | 47 |
| 3.5 | Rules Engine | 50 |
| 3.6 | Translations | 56 |
| 3.7 | Right Management | 58 |
| 3.8 | Automatic actions | 61 |
| 3.9 | Tools | 64 |
| 3.10 | Extra | 66 |
| 4 | Checklists | 69 |
| 4.1 | Review process | 69 |
| 4.2 | Prepare next major release | 69 |
| 5 | Plugins | 71 |
| 5.1 | Guidelines | 71 |
| 5.2 | Requirements | 73 |
| 5.3 | Database | 78 |
| 5.4 | Adding and managing objects | 81 |
| 5.5 | Hooks | 85 |
| 5.6 | Automatic actions | 96 |

| | | |
|----------|---|------------|
| 5.7 | Massive Actions | 96 |
| 5.8 | Tips & tricks | 98 |
| 5.9 | Notification modes | 101 |
| 5.10 | Unit Testing | 108 |
| 6 | Packaging | 111 |
| 6.1 | Sources | 111 |
| 6.2 | Filesystem Hierarchy Standard | 111 |
| 6.3 | Apache Configuration File | 112 |
| 6.4 | Logs files rotation | 113 |
| 6.5 | SELinux stuff | 114 |
| 6.6 | Use system cron | 114 |
| 6.7 | Using system libraries | 114 |
| 6.8 | Using system fonts rather than bundled ones | 115 |



SOURCE CODE MANAGEMENT

GLPI source code management is handled by [GIT](#) and hosted on [GitHub](#).

In order to contribute to the source code, you will have to know a few things about Git and the development model we follow.

1.1 Versioning

Version numbers follow the *x.y.z* nomenclature, where *x* is a major release, *y* is an intermediate release, and *z* is a bugfix release.

1.2 Backward compatibility

Wherever possible, bugfix releases should not make any non-backwards compatible changes to our source code, so a plugin that has been made compatible with a *10.0.0* release should therefore be compatible, barring exceptions, with all *10.0.x* versions. In the event that an incompatibility is introduced in a bugfix version, please let us know so that we can correct the problem.

In the context of intermediate or major versions, we do not prevent ourselves from breaking the backward compatibility of our source code. Indeed, although we try to make the maintenance of the plugins as easy as possible, some parts of our source code are not intended to be used or extended in them, and maintaining backward compatibility would be too costly in terms of time. However, the elements destined to disappear, as soon as they are intended to be used by plugins, are maintained for at least one intermediate version, and noted as being deprecated.

1.3 Branches

On the Git repository, you will find several existing branches:

- *main* (Previously named *master*) contains the next major release source code,
- *xx/bugfixes* contains the next minor release source code,
- you should not care about all other branches that may exist, they should have been deleted right now.

The *main* branch is where new features are added. This code is reputed as **non stable**.

The *x.y/bugfixes* branches is where bugs are fixed. This code is reputed as *stable*.

Those branches are created when a new major or intermediate version is released. At the time I wrote these lines, the latest stable version is *10.0* so the current bugfix branch is *10.0/bugfixes*. We do not maintain previous stable versions,

so old bugfixes branches are likely to not change; while they are still existing. In case we found a critical bug or a security issue, we may exceptionally apply patches to the latest previous stable branch.

1.4 Testing

There are more and more unit tests in GLPI; we use the [atoum unit tests framework](#).



















Every proposal **must** contains unit tests; for new features as well as bugfixes. For the bugfixes; this is not a strict requirement if this is part of code that is not tested at all yet. See the [unit testing section](#) at the bottom of the page.






























Anyways, existing unit tests may never be broken, if you made a change that breaks something, check your code, or change the unit tests, but fix that! ;)

1.5 File Hierarchy System

Note: This lists current files and directories listed in the source code of GLPI. Some files are not part of distributed archives.

This is a brief description of GLPI main folders and files:

-  `.tx`: Transifex configuration
-  `ajax`
 -  `*.php`: Ajax components
-  `files` Files written by GLPI or plugins (documents, session files, log files, ...)
-  `front`
 -  `*.php`: Front components (all displayed pages)
-  `config` (only populated once installed)
 -  `config_db.php`: Database configuration file
 -  `local_define.php`: Optional file to override some constants definitions (see `inc/define.php`)
-  `css`
 -  `...`: CSS stylesheets
 -  `*.css`: CSS stylesheets
-  `inc`
 -  `*.php`: Classes, functions and definitions
-  `install`
 -  `mysql`: MariaDB/MySQL schemas
 -  `*.php`: upgrades scripts and installer
-  `js`

-  *.js: Javascript files
-  *lib*
 -  ...: external Javascript libraries
-  *locales*
 -  *glpi.pot*: Gettext's POT file
 -  *.po: Gettext's translations
 -  *.mo: Gettext's compiled translations
-  *pics*
 -  *.*: pictures and icons
-  *plugins*:
 -  ...: where all plugins lends
-  *scripts*: various scripts which can be used in crontabs for example
-  *tests*: unit and integration tests
-  *tools*: a bunch of tools
-  *vendor*: third party libs installed from composer (see composer.json below)
-  *.gitignore*: Git ignore list
-  *.htaccess*: Some convenient apache rules (all are commented)
-  *.travis.yml*: Travis-CI configuration file
-  *apirest.php*: REST API main entry point
-  *apirest.md*: REST API documentation
-  *apixmlrpc.php*: XMLRPC API main entry point
-  *AUTHORS.txt*: list of GLPI authors
-  *CHANGELOG.md*: Changes
-  *composer.json*: Definition of third party libraries (see [composer website](#))
-  *COPYING.txt*: Licence
-  *index.php*: main application entry point
-  *phpunit.xml.dist*: unit testing configuration file
-  *README.md*: well... a README ;)
-  *status.php*: get GLPI status for monitoring purposes

1.6 Workflow

1.6.1 In short...

In a short form, here is the workflow we'll follow:

- [create a ticket](#)
- fork, create a specific branch, and hack
- open a PR (Pull Request)

Each bug will be fixed in a branch that came from the correct *bugfixes* branch. Once merged into the requested branch, developer must report the fixes in the *main*; with a simple cherry-pick for simple cases, or opening another pull request if changes are huge.

Each feature will be hacked in a branch that came from *main*, and will be merged back to *main*.

1.6.2 General

Most of the times, when you'll want to contribute to the project, you'll have to retrieve the code and change it before you can report upstream. Note that I will detail here the basic command line instructions to get things working; but of course, you'll find equivalents in your favorite Git GUI/tool/whatever ;)

Just work with a:

```
$ git clone https://github.com/glpi-project/glpi.git
```

A directory named `glpi` will be created where you've issued the clone.

Then - if you did not already - you will have to create a fork of the repository on your github account; using the *Fork* button from the [GLPI's Github page](#). This will take a few moments, and you will have a repository created, *{your user name}/glpi - forked from glpi-project/glpi*.

Add your fork as a remote from the cloned directory:

```
$ git remote add my_fork https://github.com/{your user name}/glpi.git
```

You can replace *my_fork* with what you want but *origin* (just remember it); and you will find your fork URL from the Github UI.

A basic good practice using Git is to create a branch for everything you want to do; we'll talk about that in the sections below. Just keep in mind that you will publish your branches on your fork, so you can propose your changes.

When you open a new pull request, it will be reviewed by one or more member of the community. If you're asked to make some changes, just commit again on your local branch, push it, and you're done; the pull request will be automatically updated.

Note: It's up to you to manage your fork; and keep it up to date. I'll advice you to keep original branches (such as *main* or *x.y/bugfixes*) pointing on the upstream repository.

Tha way, you'll just have to update the branch from the main repository before doing anything.

1.6.3 Bugs

If you find a bug in the current stable release, you'll have to work on the *bugfixes* branch; and, as we've said already, create a specific branch to work on. You may name your branch explicitly like *9.1/fix-something* or to reference an existing issue *9.1/fix-1234*; just prefix it with *{version}/fix-*.

Generally, the very first step for a bug is to be [filled in a ticket](#).

From the clone directory:

```
$ git checkout -b 9.1/bugfixes origin/9.1/bugfixes
$ git branch 9.1/fix-bad-api-callback
$ git co 9.1/fix-bad-api-callback
```

At this point, you're working on an only local branch named *9.1/fix-api-callback*. You can now work to solve the issue, and commit (as frequently as you want).

At the end, you will want to get your changes back to the project. So, just push the branch to your fork remote:

```
$ git push -u my_fork 9.1/fix-api-callback
```

Last step is to create a PR to get your changes back to the project. You'll find the button to do this visiting your fork or even main project github page.

Just remember here we're working on some bugfix, that should reach the *bugfixes* branch; the PR creation will probably propose you to merge against the *main* branch; and maybe will tell you they are conflicts, or many commits you do not know about... Just set the base branch to the correct bugfixes and that should be good.

1.6.4 Features

Before doing any work on any feature, make sure it has been discussed by the community. Open - if it does not exist yet - a ticket with your detailed proposition. For technical features, you can work directly on github; but for work proposals, you should take a look at our [feature proposal platform](#).

If you want to add a new feature, you will have to work on the *main* branch, and create a local branch with the name you want, prefixed with *feature/*.

From the clone directory:

```
$ git branch feature/my-killer-feature
$ git co feature/my-killer feature
```

You'll notice we do not change branch on the first step; that is just because *main* is the default branch, and therefore the one you'll be set on just after cloning. At this point, you're working on an only local branch named *feature/my-killer-feature*. You can now work and commit (as frequently as you want).

At the end, you will want to get your changes back to the project. So, just push the branch on your fork remote:

```
$ git push -u my_fork feature/my-killer-feature
```

1.6.5 Commit messages

There are several good practices regarding commit messages, but this is quite simple:

- the commit message may refer an existing ticket if any,
 - just make a simple reference to a ticket with keywords like `refs #1234` or `see #1234`",
 - automatically close a ticket when commit will be merged back with keywords like `closes #1234` or `fixes #1234`,
- the first line of the commit should be as short and as concise as possible
- if you want or have to provide details, let a blank line after the first commit line, and go on. Please avoid very long lines (some conventions talks about 80 characters maximum per line, to keep it visible).

1.6.6 Third party libraries

Third party PHP libraries are handled using the [composer tool](#) and Javascript ones using [npmjs](#).

To install existing dependencies, just install from their website or from your distribution repositories and then run:

```
$ bin/console dependencies install
```

1.7 Unit testing (and functional testing)

Note: A note for the purists... In GLPI, there are both unit and functional tests; without real distinction ;-)

We use the [atoum unit tests framework](#) for PHP tests; see [GLPI website if you wonder why](#). *atoum*'s documentation is available at: <http://docs.atoum.org>

For JavaScript tests, GLPI uses the Jest testing framework. It's documentation can be found at: <https://devdocs.io/jest/>.

Warning: With *atoum*, test class **must** refer to an existing class of the project! This means that your test class must have the same name and relative namespace as an existing class.]

1.7.1 Tests isolation

PHP tests must be run in an isolated environment. By default, *atoum* use a concurrent mode; that launches tests in a multi-threaded environment. While it is possible to bypass this; this should not be done See <http://docs.atoum.org/en/latest/engine.html>.

For technical reasons (mainly because of the huge session usage), GLPI PHP unit tests are actually limited to one only thread while running the whole suite; but while developing, the behavior should only be changed if this is really needed.

For JavaScript tests, Jest is able to run multiple tests in parallel as long as they are in different *spec* files since they don't interact with database data or a server. This behavior is the default.

1.7.2 Type hitting

Unlike PHPUnit, *atoum* is very strict on type hitting. This really makes sense; but often in GLPI types are not what we should expect (for example, we often get a string and not an integer from counting methods).

1.7.3 Database

This section is in reference to PHP tests only. JavaScript tests do not interact with a database or a GLPI server.

Each class that tests something in database **must** inherit from `\DbTestCase`. This class provides some helpers (like `login()` or `setEntity()` method); and it also does some preparation and cleanup.

Each `CommonDBTM` object added in the database with its `add()` method will be automatically deleted after the test method. If you always want to get a new object type created, you can use `beforeTestMethod()` or `setUp()` methods.

Warning: If you use `setUp()` method, do not forget to call `parent::setUp()`!

Some bootstrapped data are provided (will be inserted on the first test run); they can be used to check defaults behaviors or make queries, but you should **never change those data!** This lead to unpredictable further tests results.

1.7.4 Variables declaration

When you use a property that has not been declared, you will have errors that may be quite difficult to understand. Just remember to always declare property you use!

```
<?php

class MyClass extends atoum {
    private $myprop;

    public function testMethod() {
        $this->myprop = 'foo'; //<-- error here if missing "private $myprop"
    }
}
```

1.7.5 Launch tests

You can install *atoum* from composer (just run `composer install` from GLPI directory) or even system wide.

There are two directories for tests:

- `tests/units` for main core tests;
- `tests/api` for API tests.

You can choose to run tests on a whole directory, or on any file (+ on a specific method). You have to specify a bootstrap file each time:

```
$ atoum -bf tests/bootstrap.php -mcn 1 -d tests/units/
[...]
$ atoum -bf tests/bootstrap.php -f tests/units/Html.php
[...]
```

(continues on next page)

(continued from previous page)

```
$ atoum -bf tests/bootstrap.php -f tests/functional/Ticket.php -m tests\units\  
↪Ticket::testTechAcls
```

If you want to run the API tests suite, you need to run a development server:

```
php -S localhost:8088 tests/router.php &>/dev/null &
```

Running *atoum* without any arguments will show you the possible options. Most important are:

- `-bf` to set bootstrap file,
- `-d` to run tests located in a whole directory,
- `-f` to run tests on a standalone file,
- `-m` to run tests on a specific method (`-f` must also be defined),
- `--debug` to get extra information when something goes wrong,
- `-mcn` limit number of concurrent runs. This is unfortunately mandatory running the whole test suite right now :/,
- `-ncc` do not generate code coverage,
- `--php` to change PHP executable to use,
- `-l` loop mode.

Note that if you do not use the `-ncc` switch; coverage will be generated in the `tests/code-coverage/` directory.

To run the JavaScript unit tests, simply run `npm test` in a terminal from the root of the GLPI folder. Currently, there is only a single “project” set up for Jest so this command will run all tests.



CODING STANDARDS

As of GLPI 10, we rely on [PSR-12](#) for coding standards.

2.1 Call static methods

| Function location | How to call |
|-------------------|-------------------------------------|
| class itself | <code>self::theMethod()</code> |
| parent class | <code>parent::theMethod()</code> |
| another class | <code>ClassName::theMethod()</code> |

2.2 Static or Non static?

Some methods in the source code are [declared as static](#); some are not.

For sure, you cannot make static calls on a non static method. In order to call such a method, you will have to get an object instance, and then call the method on it:

```
<?php
$object = new MyObject();
$object->nonStaticMethod();
```

It may be different calling static classes. In that case; you can either:

- call statically the method from the object; like `MyObject::staticMethod()`,
- call statically the method from an object instance; like `$object::staticMethod()`,
- call non statically the method from an object instance; like `$object->staticMethod()`.
- use [late static building](#); like `static::staticMethod()`.

When you do not have any object instance yet; the first solution is probably the best one. No need to instantiate an object to just call a static method from it.

On the other hand; if you already have an object instance; you should better use any of the solution but the late static binding. That way; you will save performances since this way to go do have a cost.

2.3 Comments

To be more visible, don't put inline block comments into `/* */` but comment each line with `//`. Put docblocks comments into `/** */`.

Each function or method must be documented, as well as all its parameters (see *Variables types* below), and its return.

For each method or function documentation, you'll need at least to have a description, the version it was introduced, the parameters list, the return type; each blocks separated with a blank line. As an example, for a void function:

```
<?php
/**
 * Describe what the method does. Be concise :)
 *
 * You may want to add some more words about what the function
 * does, if needed. This is optional, but you can be more
 * descriptive here:
 * - it does something
 * - and also something else
 * - but it doesn't make coffee, unfortunately.
 *
 * @since 9.2
 *
 * @param string $param      A parameter, for something
 * @param boolean $other_param Another parameter
 *
 * @return void
 */
function myMethod($param, $other_param) {
    //[...]
}
```

Some other information way be added; if the function requires it.

Refer to the [PHPDocumentor website](#) to get more information on documentation.

Please follow the order defined below:

1. Description,
2. Long description, if any,
3. `@deprecated`.
4. `@since`,
5. `@var`,
6. `@param`,
7. `@return`,
8. `@see`,
9. `@throw`,
10. `@todo`,

2.3.1 Parameters documentation

Each parameter must be documented in its own line, beginning with the `@param` tag, followed by the *Variables types*, followed by the param name (`$param`), and finally with the description itself. If your parameter can be of different types, you can list them separated with a `|` or you can use the `mixed` type; it's up to you!

All parameters names and description must be aligned vertically on the longest (plu one character); see the above example.

2.3.2 Override method: `@inheritDoc?` `@see?` `docblock?` `no docblock?`

There are many question regarding the way to document a child method in a child class.

Many editors use the `{@inheritDoc}` tag without anything else. **This is wrong**. This *inline* tag is confusing for many users; for more details, see the [PHPDocumentor documentation about it](#). This tag usage is not forbidden, but make sure to use it properly, or just avoid it. An usage example:

```
<?php

abstract class MyClass {
    /**
     * This is the documentation block for the current method.
     * It does something.
     *
     * @param string $stthing Something to send to the method
     *
     * @return string
     */
    abstract public function myMethod($stthing);
}

class MyChildClass extends MyClass {
    /**
     * {@inheritDoc} Something is done differently for a reason.
     *
     * @param string $stthing Something to send to the method
     *
     * @return string
     */
    public function myMethod($stthing) {
        [...]
    }
}
```

Something we can see quite often is just the usage of the `@see` tag to make reference to the parent method. **This is wrong**. The `@see` tag is designed to reference another method that would help to understand this one; not to make a reference to its parent (you can also take a look at [PHPDocumentor documentation about it](#)). While generating, parent class and methods are automatically discovered; a link to the parent will be automatically added. An usage example:

```
<?php
/**
 * Adds something
 *
 * @param string $type Type of thing
 * @param string $value The value
```

(continues on next page)

(continued from previous page)

```

*
* @return boolean
*/
public function add($type, $value) {
    // [...]
}

/**
 * Adds myType entry
 *
 * @param string $value The value
 *
 * @return boolean
 * @see add()
 */
public function addMyType($value) {
    return $this->addType('myType', $value);
}

```

Finally, should I add a docblock, or nothing?

PHPDocumentor and various tools will just use parent docblock verbatim if nothing is specified on child methods. So, if the child method acts just as its parent (extending an abstract class, or some super class like `CommonGLPI` or `CommonDBTM`); you may just omit the docblock entirely. The alternative is to copy paste parent docblock entirely; but that way, it would be required to change all children docblocks when parent if changed.

2.4 Variables types

Variables types for use in DocBlocks for Doxygen:

| Type | Description |
|----------|--|
| mixed | A variable with undefined (or multiple) type |
| integer | Integer type variable (whole number) |
| float | Float type (point number) |
| boolean | Logical type (true or false) |
| string | String type (any value in "" or ' ') |
| array | Array type |
| object | Object type |
| resource | Resource type (as returned from <code>mysql_connect</code> function) |

In addition to the above, you may use any valid types from [PHPStan](#).

You may also use a specific class for the type as a replacement for *object* when you know the exact type of data being used. This is recommended if you use typehints. Since PHP 7.1, you can have nullable typehints for method parameters and return types. You should prepend a `?` to the above types if they are nullable.

Inserting comment in source code for doxygen. Result : full doc for variables, functions, classes...

2.5 Quotes / double quotes

- You must use single quotes for indexes, constants declaration, translations, ...
- Use double quote in translated strings
- When you have to use tabulation character (`\t`), carriage return (`\n`) and so on, you should use double quotes.
- For performances reasons since PHP7, you may avoid strings concatenation.

Examples:

```
<?php
//for that one, you should use double, but this is at your option...
$a = "foo";

//use double quotes here, for $foo to be interpreted
//  => with double quotes, $a will be "Hello bar" if $foo = 'bar'
//  => with single quotes, $a will be "Hello $foo"
$a = "Hello $foo";

//use single quotes for array keys
$tab = [
    'lastname' => 'john',
    'firstname' => 'doe'
];

//Do not use concatenation to optimize PHP7
//note that you cannot use functions call in {}
$a = "Hello {$tab['firstname']}";

//single quote translations
$str = __('My string to translate');

//Double quote for special characters
$html = "<p>One paragraph</p>\n<p>Another one</p>";

//single quote cases
switch ($a) {
    case 'foo' : //use single quote here
        ...
    case 'bar' :
        ...
}
```

2.6 Checking standards

In order to check standards are respected, we provide a default configuration for [PHP CodeSniffer](#) rules. From the GLPI directory, just run:

```
phpcs .
```

If the above command does not provide any output, then, all is OK :)

An example error output would looks like:

```
phpcs .
```

```
FILE: /var/www/webapps/glpi/tests/HtmlTest.php
```

```
-----  
FOUND 3 ERRORS AFFECTING 3 LINES  
-----
```

```
40 | ERROR | [x] Line indented incorrectly; expected 4 spaces, found  
   |       | 3  
59 | ERROR | [x] Line indented incorrectly; expected 4 spaces, found  
   |       | 3  
64 | ERROR | [x] Line indented incorrectly; expected 4 spaces, found  
   |       | 3
```

To automatically fix most of the issues, use *phpcbf*, it will per default rely on default configuration:

```
phpcbf .
```



DEVELOPER API

Apart from the current documentation, you can also generate the full PHP documentation of GLPI (built with [apigen](#)) using the `tools/genapidoc.sh` script.

3.1 Main framework objects

GLPI contains numerous classes; but there are a few common objects you'd have to know about. All GLPI classes are in the `src` directory. Prior to GLPI 10.0, the classes were in the `inc` directory. Now, only non-class PHP files remain there.

Note: See the full API documentation for related object for a complete list of methods provided.

3.1.1 CommonGLPI

This is **the** main GLPI object, most of GLPI or Plugins class inherit from this one, directly or not.

This object will help you to:

- manage item type name,
- manage item tabs,
- manage item menu,
- do some display,
- get URLs (form, search, ...),
- ...

3.1.2 CommonDBTM

This is an object to manage any database stuff; it of course inherits from *CommonGLPI*.

It aims to manage database persistence and tables for all objects; and will help you to:

- add, update or delete database rows,
- load a row from the database,
- get table informations (name, indexes, relations, ...)
- ...

The CommonDBTM object provides several of the *available hooks*.

3.1.3 CommonDropdown

This class aims to manage dropdown (lists) database stuff. It inherits from *CommonDBTM*.

It will help you to:

- manage the list,
- import data,
- ...

3.1.4 CommonTreeDropdown

This class aims to manage tree lists database stuff. It inherits from *CommonDropdown*.

It will mainly help you to manage the tree aspect of a dropdown (parents, children, and so on).

3.1.5 CommonImplicitTreeDropdown

This class manages tree lists that cannot be managed by the user. It inherits from *CommonTreeDropdown*.

3.1.6 CommonDBVisible

This class helps with visibility management. It inherits from *CommonDBTM*.

It provides methods to:

- know if the user can view item,
- get dropdown parameters,
- ...

3.1.7 CommonDBConnexity

This class factorizes database relation and inheritance stuff. It inherits from *CommonDBTM*.

It is not designed to be used directly, see *CommonDBChild* and *CommonDBRelation*.

3.1.8 CommonDBChild

This class manages simple relations. It inherits from *CommonDBConnexity*.

This object will help you to define and manage parent/child relations.

3.1.9 CommonDBRelation

This class manages relations. It inherits from *CommonDBConnexity*.

Unlike *CommonDBChild*; it is designed to declare more *complex relations; as defined in the database model*. This is therefore more complex than just using a simple relation; but it also offers many more possibilities.

In order to setup a complex relation, you'll have to define several properties, such as:

- `$itemtype_1` and `$itemtype_2`; to set both itm types used;
- `$items_id_1` and `$items_id_2`; to set field id name.

Other properties let you configure how to deal with entities inheritance, ACLs; what to log on each part on several actions, and so on.

The object will also help you to:

- get search options and query,
- find rights in ACLs list,
- handle massive actions,
- ...

3.1.10 CommonDevice

This class factorizes common requirements on devices. It inherits from *CommonDropdown*.

It will help you to:

- import devices,
- handle menus,
- do some display,
- ...

3.1.11 Common ITIL objects

All common ITIL objects will help you with **ITIL** objects management (Tickets, Changes, Problems).

CommonITILObject

Handle ITIL objects. It inherits from *CommonDBTM*.

It will help you to:

- get users, suppliers, groups, ...
- count them,
- get objects for users, technicians, suppliers, ...
- get status,
- ...

CommonITILActor

Handle ITIL actors. It inherits from *CommonDBRelation*.

It will help you to:

- get actors,
- show notifications,
- get ACLs,
- ...

CommonITILCost

Handle ITIL costs. It inherits from *CommonDBChild*.

It will help you to:

- get item cost,
- do some display,
- ...

CommonITILTask

Handle ITIL tasks. It inherits from *CommonDBTM*.

It will help you to:

- manage tasks ACLs,
- do some display,
- get search options,
- ...

CommonITILValidation

Handle ITIL validation process. It inherits from *CommonDBChild*.

It will help you to:

- mange ACLs,
- get and set status,
- get counts,
- do some display,
- ...

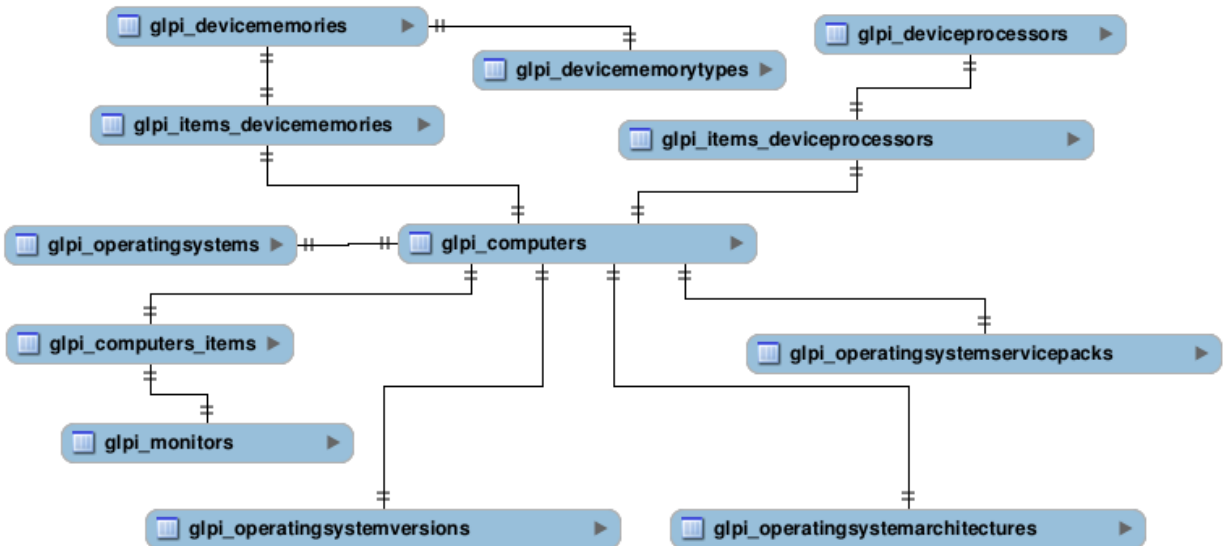


3.2 Database

3.2.1 Database model

Current GLPI database contains more than 250 tables; the goal of the current documentation is to help you to understand the logic of the project, not to detail each table and possibility.

As on every database, there are tables, relations between them (more or less complex), some relations have descriptions stored in a another table, some tables way be linked with themselves... Well, it's quite common :) Let's start with a simple example:



Note: The above schema is an example, it is far from complete!

What we can see here:

- computers are directly linked to operating systems, operating systems versions, operating systems architectures, ...,
- computers are linked to memories, processors and monitors using a relation table (which in that case permit to link those components to other items than a computer),
- memories have a type.

As stated in the above note, this is far from complete; but this is quite representative of the whole database schema.

Resultsets

All resultsets sent back from GLPI database should always be associative arrays.

Naming conventions

All tables and fields names are lower case and follows the same logic. If you do not respect that; GLPI will fail to find relevant information.

Tables

Tables names are linked with PHP classes names; they are all prefixed with `glpi_`, and class name is set to plural. Plugins tables must be prefixed by `glpi_plugin_`; followed by the plugin name, another dash, and then pluralized class name.

A few examples:

| PHP class name | Table name |
|----------------------|---|
| Computer | <code>glpi_computers</code> |
| Ticket | <code>glpi_tickets</code> |
| ITILCategory | <code>glpi_itilcategories</code> |
| PluginExampleProfile | <code>glpi_plugin_example_profiles</code> |

Fields

Warning: Each table **must** have an auto-incremented primary key named `id`.

Field naming is mostly up to you; except for identifiers and foreign keys. Just keep clear and concise!

To add a foreign key field; just use the foreign table name without `glpi_` prefix, and add `_id` suffix.

Warning: Even if adding a foreign key in a table should be perfectly correct; this is not the usual way things are done in GLPI, see [Make relations](#) to know more.

A few examples:

| Table name | Foreign key field name |
|---|---|
| <code>glpi_computers</code> | <code>computers_id</code> |
| <code>glpi_tickets</code> | <code>tickets_id</code> |
| <code>glpi_itilcategories</code> | <code>itilcategories_id</code> |
| <code>glpi_plugin_example_profiles</code> | <code>plugin_example_profiles_id</code> |

Make relations

On most cases, you may want to made possible to link many different items to something else. Let's say you want to make possible to link a *Computer*, a *Printer* or a *Phone* to a *Memory* component. You should add foreign keys in items tables; but on something as huge as GLPI, it maybe not a good idea.

Instead, create a relation table, that will reference the memory component along with a item id and a type, as for example:

```
CREATE TABLE `glpi_items_devicememories` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `items_id` int(11) NOT NULL DEFAULT '0',
  `itemtype` varchar(255) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `devicememories_id` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`),
  KEY `items_id` (`items_id`),
  KEY `devicememories_id` (`devicememories_id`),
  KEY `itemtype` (`itemtype`,`items_id`),
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci ROW_FORMAT=DYNAMIC;
```

Again, this is a very simplified example of what already exists in the database, but you got the point ;)

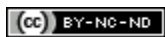
In this example, itemtype would be Computer, Printer or Phone; items_id the id of the related item.

Indexes

In order to get correct performances querying database, you'll have to take care of setting some indexes. It's a nonsense to add indexes on every fields in the database; but some of them must be defined:

- foreign key fields;
- fields that are very often used (for example fields like `is_visible`, `itemtype`, ...),
- primary keys ;)

You should just use the field name as key name.



3.2.2 Querying

GLPI framework provides a simple request generator:

- without having to write SQL
- without having to quote table and field name
- without having to take care of freeing resources
- iterable
- countable

Basic usage

```
<?php
foreach ($DB->request(...) as $id => $row) {
    //... work on each row ...
}

$req = $DB->request(...);
if ($row = $req->next()) {
    // ... work on a single row
}

$req = $DB->request(...);
if (count($req)) {
    // ... work on result
}
```

Arguments

The `request` method takes two arguments:

- *table name(s)*: a *string* or an *array* of *string* (optional when given as `FROM` option)
- *option(s)*: array of options

Giving full SQL statement

If the only option is a full SQL statement, it will be used. This usage is deprecated, and should be avoid when possible.

Note: To make a database query that could not be done using recommended way (calling SQL functions such as `NOW()`, `ADD_DATE()`, ... for example), you can do:

```
<?php
$DB->request('SHOW COLUMNS FROM `glpi_computers`');
```

Without option

In this case, all the data from the selected table is iterated:

```
<?php
$DB->request(['FROM' => 'glpi_computers']);
// => SELECT * FROM `glpi_computers`

$DB->request('glpi_computers');
// => SELECT * FROM `glpi_computers`
```

Fields selection

You can use either the SELECT or FIELDS options, an additional DISTINCT option might be specified.

Note: Changed in version 9.5.0.

Using DISTINCT FIELDS or SELECT DISTINCT options is deprecated.

```
<?php
$DB->request(['SELECT' => 'id', 'FROM' => 'glpi_computers']);
// => SELECT `id` FROM `glpi_computers`

$DB->request(['FIELDS' => 'id', 'FROM' => 'glpi_computers']);
// => SELECT `id` FROM `glpi_computers`

$DB->request(['SELECT' => 'name', 'DISTINCT' => true, 'FROM' => 'glpi_computers']);
// => SELECT DISTINCT `name` FROM `glpi_computers`

$DB->request(['FIELDS' => 'name', 'DISTINCT' => true, 'FROM' => 'glpi_computers']);
// => SELECT DISTINCT `name` FROM `glpi_computers`
```

The fields array can also contain per table sub-array:

```
<?php
$DB->request(['FIELDS' => ['glpi_computers' => ['id', 'name']], 'FROM' => 'glpi_computers'
// => SELECT `glpi_computers`.`id`, `glpi_computers`.`name` FROM `glpi_computers`
```

Using JOINS

You need to use criteria, usually a FKEY to describe how to join the tables.

Note: New in version 9.3.1.

The ON keyword can also be used as an alias of FKEY.

Multiple tables, native join

You need to use criteria, usually a FKEY (or the ON equivalent), to describe how to join the tables:

```
<?php
$DB->request(['FROM' => ['glpi_computers', 'glpi_computerdisks'],
                'FKEY' => ['glpi_computers'=>'id',
                          'glpi_computerdisks'=>'computer_id']]);
$DB->request(['glpi_computers', 'glpi_computerdisks'],
                ['FKEY' => ['glpi_computers'=>'id',
                          'glpi_computerdisks'=>'computer_id']]);
// => SELECT * FROM `glpi_computers`, `glpi_computerdisks`
//      WHERE `glpi_computers`.`id` = `glpi_computerdisks`.`computer_id`
```

Left join

Using the `LEFT JOIN` option, with some criteria, usually a FKEY (or the ON equivalent):

```
<?php
$DB->request(['FROM'          => 'glpi_computers',
              'LEFT JOIN' => ['glpi_computerdisks' => ['FKEY' => ['glpi_computers' =>
↳ 'id',
                                                                    'glpi_computerdisks' =>
↳ 'computer_id']]]]);
// => SELECT * FROM `glpi_computers`
//      LEFT JOIN `glpi_computerdisks`
//      ON (`glpi_computers`.`id` = `glpi_computerdisks`.`computer_id`)
```

Inner join

Using the `INNER JOIN` option, with some criteria, usually a FKEY (or the ON equivalent):

```
<?php
$DB->request(['FROM'          => 'glpi_computers',
              'INNER JOIN' => ['glpi_computerdisks' => ['FKEY' => ['glpi_computers' =>
↳ 'id',
                                                                    'glpi_computerdisks' =>
↳ 'computer_id']]]]);
// => SELECT * FROM `glpi_computers`
//      INNER JOIN `glpi_computerdisks`
//      ON (`glpi_computers`.`id` = `glpi_computerdisks`.`computer_id`)
```

Right join

Using the `RIGHT JOIN` option, with some criteria, usually a FKEY (or the ON equivalent):

```
<?php
$DB->request(['FROM'          => 'glpi_computers',
              'RIGHT JOIN' => ['glpi_computerdisks' => ['FKEY' => ['glpi_computers' =>
↳ 'id',
                                                                    'glpi_computerdisks' =>
↳ 'computer_id']]]]);
// => SELECT * FROM `glpi_computers`
//      RIGHT JOIN `glpi_computerdisks`
//      ON (`glpi_computers`.`id` = `glpi_computerdisks`.`computer_id`)
```

Join criterion

New in version 9.3.1.

It is also possible to add an extra criterion for any *JOIN* clause. You have to pass an array with first key equal to AND or OR and any iterator valid criterion:

```
<?php
$DB->request([
    'FROM' => 'glpi_computers',
    'INNER JOIN' => [
        'glpi_computerdisks' => [
            'FKEY' => [
                'glpi_computers' => 'id',
                'glpi_computerdisks' => 'computer_id',
                ['OR' => ['glpi_computers.field' => ['>', 42]]]
            ]
        ]
    ]
]);

// => SELECT * FROM `glpi_computers`
//      INNER JOIN `glpi_computerdisks`
//      ON (`glpi_computers`.`id` = `glpi_computerdisks`.`computer_id` OR
//          `glpi_computers`.`field` > '42'
//      )
```

UNION queries

New in version 9.4.0.

An union query is an object, which contains an array of *Sub queries*. You just have to give a list of Subqueries you have already prepared, or arrays of parameters that will be used to build them.

```
<?php
$sub1 = new \QuerySubQuery([
    'SELECT' => 'field1 AS myfield',
    'FROM' => 'table1'
]);
$sub2 = new \QuerySubQuery([
    'SELECT' => 'field2 AS myfield',
    'FROM' => 'table2'
]);
$union = new \QueryUnion([$sub1, $sub2]);
$DB->request([
    'FROM' => $union
]);

// => SELECT * FROM (
//      SELECT `field1` AS `myfield` FROM `table1`
//      UNION ALL
//      SELECT `field2` AS `myfield` FROM `table2`
//      )
```

As you can see on the above example, a UNION ALL query is built. If you want your results to be deduplicated, (standard UNION):

```
<?php
//...
//passing true as second argument will activate deduplication.
$union = new \QueryUnion([$sub1, $sub2], true);
//...
```

Warning: Keep in mind that deduplicating a UNION query may have a huge cost on database server.

Most of the time, you can issue a UNION ALL and deduplicate the results in the code.

Counting

Using the COUNT option:

```
<?php
$DB->request(['FROM' => 'glpi_computers', 'COUNT' => 'cpt']);
// => SELECT COUNT(*) AS cpt FROM `glpi_computers`
```

Grouping

Using the GROUPBY option, which contains a field name or an array of field names.

```
<?php
$DB->request(['FROM' => 'glpi_computers', 'GROUPBY' => 'name']);
// => SELECT * FROM `glpi_computers` GROUP BY `name`

$DB->request('glpi_computers', ['GROUPBY' => ['name', 'states_id']]);
// => SELECT * FROM `glpi_computers` GROUP BY `name`, `states_id`
```

Order

Using the ORDER option, with value a field or an array of fields. Field name can also contains ASC or DESC suffix.

```
<?php
$DB->request(['FROM' => 'glpi_computers', 'ORDER' => 'name']);
// => SELECT * FROM `glpi_computers` ORDER BY `name`

$DB->request('glpi_computers', ['ORDER' => ['date_mod DESC', 'name ASC']]);
// => SELECT * FROM `glpi_computers` ORDER BY `date_mod` DESC, `name` ASC
```


Request pager

Using the START and LIMIT options:

```
<?php
$DB->request('glpi_computers', ['START' => 5, 'LIMIT' => 10]);
// => SELECT * FROM `glpi_computers` LIMIT 10 OFFSET 5"
```

Criteria

Using the WHERE option with an array of criteria. The first level of the array is considered as an implicit logical AND. By default, the array keys are considered as field names, and the values as values. If this differs from what you want, there are a few workarounds that are covered later.

Simple criteria

A field name and its wanted value:

```
<?php
$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['is_deleted' => 0]]);
// => SELECT * FROM `glpi_computers` WHERE `is_deleted` = 0

$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['is_deleted' => 0,
                                                    'name' => 'foo']]);
// => SELECT * FROM `glpi_computers` WHERE `is_deleted` = 0 AND `name` = 'foo'

$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['users_id' => [1,5,7]]]);
// => SELECT * FROM `glpi_computers` WHERE `users_id` IN (1, 5, 7)
```

When using an array as a value, the operator is automatically set to IN. Make sure that you verify that the array cannot be empty, otherwise an error will be thrown.

When using null as a value, the operator is automatically set to IS and the value is set to the NULL keyword.

Logical OR, AND, NOT

Using the OR, AND, or NOT option with an array of criteria:

```
<?php
$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['OR' => ['is_deleted' => 0,
                                                            'name' => 'foo']]]);
// => SELECT * FROM `glpi_computers` WHERE (`is_deleted` = 0 OR `name` = 'foo')

$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['NOT' => ['id' => [1,2,7]]]]);
// => SELECT * FROM `glpi_computers` WHERE NOT (`id` IN (1, 2, 7))
```

Using a more complex expression with AND and OR:

```
<?php
$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['is_deleted' => 0,
                                                    ['OR' => ['name' => 'foo', 'otherserial' => 'otherunique']],
```

(continues on next page)

(continued from previous page)

```

    ['OR' => ['locations_id' => 1, 'serial' => 'unique']]
  ]];
// => SELECT * FROM `glpi_computers` WHERE `is_deleted` = '0' AND ((`name` = 'foo' OR
↳ `otherserial` = 'otherunique')) AND ((`locations_id` = '1' OR `serial` = 'unique'))

```

Operators

Default operator is =, but other operators can be used, by giving an array containing operator and value.

```

<?php
$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['date_mod' => ['>' , '2016-10-01'
↳ ']]]);
// => SELECT * FROM `glpi_computers` WHERE `date_mod` > '2016-10-01'

$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['name' => ['LIKE' , 'pc00%']]]);
// => SELECT * FROM `glpi_computers` WHERE `name` LIKE 'pc00%'

```

Known operators are =, !=, <, <=, >, >=, LIKE, REGEXP, NOT LIKE, NOT REGEX, & (BITWISE AND), and | (BITWISE OR).

Aliases

You can use SQL aliases (SQL AS keyword). To achieve that, just write the alias you want on the table name or the field name; then use it in your parameters:

```

<?php
$DB->request(['FROM' => 'glpi_computers AS c']);
// => SELECT * FROM `glpi_computers` AS `c`

$DB->request(['SELECT' => 'field AS f', 'FROM' => 'glpi_computers AS c']);
// => SELECT `field` AS `f` FROM `glpi_computers` AS `c`

```

Aggregate functions

New in version 9.3.1.

You can use some aggregation SQL functions on fields: COUNT, SUM, AVG, MIN and MAX are supported. Just set the function as the key in your fields array:

```

<?php
$DB->request(['SELECT' => ['COUNT' => 'field', 'bar'], 'FROM' => 'glpi_computers',
↳ 'GROUPBY' => 'field']);
// => SELECT COUNT(`field`), `bar` FROM `glpi_computers` GROUP BY `field`

$DB->request(['SELECT' => ['bar', 'SUM' => 'amount AS total'], 'FROM' => 'glpi_computers
↳ ', 'GROUPBY' => 'amount']);
// => SELECT `bar`, SUM(`amount`) AS `total` FROM `glpi_computers` GROUP BY `amount`

```

Sub queries

New in version 9.3.1.

You can use subqueries, using the specific *QuerySubQuery* class. It takes two arguments: the first is an array of criteria to get the query built, and the second is an optional operator to use. Allowed operators are the same than documented below plus *IN* and *NOT IN*. Default operator is *IN*.

```
<?php
$sub_query = new \QuerySubQuery([
    'SELECT' => 'id',
    'FROM'    => 'subtable',
    'WHERE'   => [
        'subfield' => 'subvalue'
    ]
]);
$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['field' => $sub_query]]);
// => SELECT * FROM `glpi_computers` WHERE `field` IN (SELECT `id` FROM `subtable` WHERE
↳ `subfield` = 'subvalue')

$sub_query = new \QuerySubQuery([
    'SELECT' => 'id',
    'FROM'    => 'subtable',
    'WHERE'   => [
        'subfield' => 'subvalue'
    ]
]);
$DB->request(['FROM' => 'glpi_computers', 'WHERE' => ['NOT' => ['field' => $sub_
↳ query]]]);
// => SELECT * FROM `glpi_computers` WHERE NOT `field` IN (SELECT `id` FROM `subtable`
↳ WHERE `subfield` = 'subvalue')

$sub_query = new \QuerySubQuery([
    'SELECT' => 'id',
    'FROM'    => 'subtable',
    'WHERE'   => [
        'subfield' => 'subvalue'
    ]
], 'myalias');
$DB->request(['FROM' => 'glpi_computers', 'SELECT' => [$sub_query, 'id']]);
// => SELECT (SELECT `id` FROM `subtable` WHERE `subfield` = 'subvalue') AS `myalias`, id
↳ FROM `glpi_computers`
```

What if iterator does not provide what I'm looking for?

Even if we do our best to get as many things as possible implemented in the iterator, there are several things that are missing... Consider for example you want to use the SQL *NOW()* function, or want to use a value based on another field: there is no native way to achieve that.

Right now, there is a *QueryExpression* class that would permit to do such things on values (an not on fields since it is not possible to use a class instance as an array key).

Warning: The *QueryExpression* class will pass raw SQL. You are in charge to escape name and values you use into it!

For example, to use the SQL *NOW()* function:

```
<?php
$DB->request([
    'FROM'    => 'my_table',
    'WHERE'   => [
        'date_end' => ['>', new QueryExpression('NOW()')]
    ]
]);
// SELECT * FROM `my_table` WHERE `date_end` > NOW()
```

An example with a field value:

```
<?php
$DB->request([
    'FROM'    => 'my_table',
    'WHERE'   => [
        'field' => new QueryExpression(DBmysql::quoteName('other_field'))
    ]
]);
// SELECT * FROM `my_table` WHERE `field` = `other_field`
```

New in version 9.3.1.

You can also use some function or non supported stuff on field part by using a *RAW* entry in the query:

```
<?php
$DB->request([
    'FROM'    => 'my_table',
    'WHERE'   => [
        'RAW'  => [
            DBmysql::quoteName('field') => DBmysql::quoteName('field2')
        ]
    ]
]);
// SELECT * FROM `my_table` WHERE LOWER(`field`) = 'value'
```

New in version 9.5.0.

You can use a *QueryExpression* object in the *FIELDS* statement:

```
<?php
$DB->request([
    'FIELDS'   => [
        'glpi_computers' => ['id'],
        new QueryExpression("CONCAT(`glpi_computers`.`name`, ' ', `glpi_domains`.`name`)")
        ↪ AS `fullname`"
    ],
    'FROM'     => 'glpi_computers',
    'LEFT JOIN' => [
        'glpi_domains' => [
```

(continues on next page)

(continued from previous page)

```

        'FKEY' => [
            'glpi_computers' => 'domains_id',
            'glpi_domains' => 'id',
        ]
    ]
}
];
// => SELECT `glpi_computers`.`id`, CONCAT(`glpi_computers`.`name`, '.', `glpi_domains`.`
↳ `name`) AS `fullname` FROM `glpi_computers` LEFT JOIN `glpi_domains` ON (`glpi_computers`.`
↳ `domains_id` = `glpi_domains`.`id`)

```

You can use a QueryExpression object in the FROM statement:

```

<?php
$DB->request([
    'FROM'      => new QueryExpression('(SELECT * FROM glpi_computers) as computers'),
]);
// => SELECT * FROM (SELECT * FROM glpi_computers) as computers

```

When you need to manually quote identifies or values, it is recommended that you use `$DB::quoteName` and `$DB::quoteValue` respectively rather than directly adding the quotes to ensure future compatibility.



3.2.3 Updating

New in version 9.3.

Just as SQL *SELECT* queries, you should avoid plain SQL and use methods provided by the framework from the DB object.

General

Escaping of data is currently provided automatically by the framework for all data passed from *GET* or *POST*; you do not have to take care of them (this will change in a future version). You have to take care of escaping data when you use values that came from elsewhere.

The *WHERE* part of *UPDATE* and *DELETE* methods uses the same *criteria capabilities* than *SELECT* queries.

Inserting a row

You can insert a row in the database using the `insert()`:

```

<?php
$DB->insert(
    'glpi_my_table', [
        'a_field'      => 'My value',
        'other_field' => 'Other value'
    ]
);

```

(continues on next page)

(continued from previous page)

```
// => INSERT INTO `glpi_my_table` (`a_field`, `other_field`) VALUES ('My value', Other_
↪value)
```

An insertOrDie() method is also provided.

Updating a row

You can update rows in the database using the update() method:

```
<?php
$DB->update(
    'glpi_my_table', [
        'a_field'      => 'My value',
        'other_field' => 'Other value'
    ], [
        'id' => 42
    ]
);
// => UPDATE `glpi_my_table` SET `a_field` = 'My value', `other_field` = 'Other value' WHERE_
↪`id` = 42
```

An updateOrDie() method is also provided.

New in version 9.3.1.

When issuing an *UPDATE* query, you can use an *ORDER* and/or a *LIMIT* clause along with the where (which remains **mandatory**). In order to achieve that, use an indexed array with appropriate keys:

```
<?php
$DB->update(
    'my_table', [
        'my_field' => 'my value'
    ], [
        'WHERE' => ['field' => 'value'],
        'ORDER' => ['date DESC', 'id ASC'],
        'LIMIT' => 1
    ]
);
```

Removing a row

You can remove rows from the database using the delete() method:

```
<?php
$DB->delete(
    'glpi_my_table', [
        'id' => 42
    ]
);
// => DELETE FROM `glpi_my_table` WHERE `id` = 42
```

Use prepared statements

On some cases, you may want to use prepared statements to improve performances. In order to achieve that, you will have to create a query with some parameters (not named, since mysqli does not supports named parameters), then to prepare it, and finally to bind parameters and execute the statement.

Let's see an example with an insert statement:

```
<?php
$insert_query = $DB->buildInsert(
    'my_table', [
        'field' => new QueryParam(),
        'other' => new QueryParam()
    ]
);
// => INSERT INTO `glpi_my_table` (`field`, `other`) VALUES (?, ?)
$stmt = $DB->prepare($insert_query);

foreach ($data as $row) {
    $stmt->bind_params(
        'ss',
        $row['field'],
        $row['other']
    );
    $stmt->execute();
}
```

Just like the *buildInsert()* method used here, *buildUpdate* and *buildDelete* methods are available. They take exactly the same arguments as “non build” methods.

Note: Note the use of the *QueryParam* object. This is used for the builder to be aware you are not passing a value, but a parameter (that must not be escaped nor quoted).

Preparing a *SELECT* query is a bit different:

```
<?php
$it = new DBmysqlIterator();
$it->buildQuery([
    'FROM' => 'my_table',
    'WHERE' => [
        'something' => new QueryParam(),
        'foo' => 'bar'
    ]
]);
$query = $it->getSql();
// => SELECT FROM `my_table` WHERE `something` = ? AND `foo` = 'bar'
$stmt = $DB->prepare($query);
// [...]
```



3.3 Search Engine

3.3.1 Goal

The Search class aims to provide a multi-criteria Search engine for GLPI Itemtypes.

It includes some short-cuts functions:

- `show()`: displays the complete search page.
- `showGenericSearch()`: displays only the multi-criteria form.
- `showList()`: displays only the resulting list.
- `getDdatas()`: return an array of raw data.
- `manageParams()`: complete the `$_GET` values with the `$_SESSION` values.

The show function parse the `$_GET` values (calling `manageParams()`) passed by the page to retrieve the criteria and construct the SQL query. For `showList` function, *parameters* can be passed in the second argument.

The itemtype classes can define a set of *search options* to configure which columns could be queried, how they can be accessed and displayed, etc..

Todo:

- datafields option
 - difference between searchunit and delay_unit
 - dropdown translations
 - giveItem
 - export
 - fulltext search
-

Examples

To display the search engine with its default options (criteria form, pager, list):

```
<?php
$itemtype = 'Computer';
Search::show($itemtype);
```

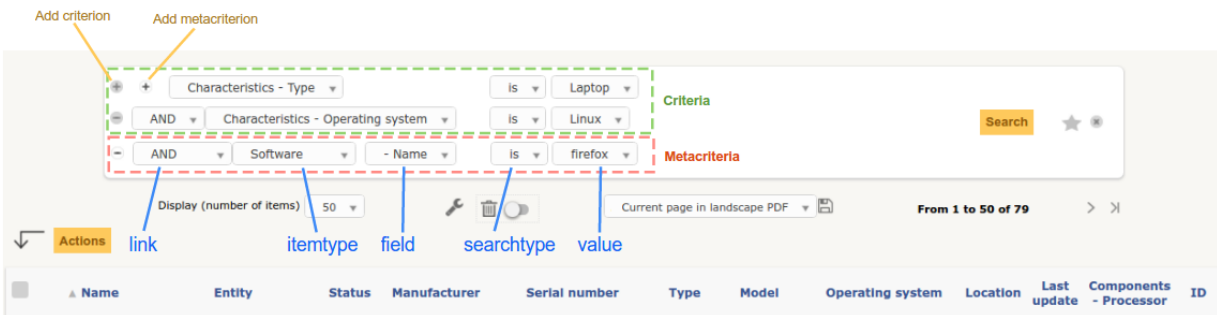
If you want to display only the multi-criteria form (with some additional options):

```
<?php
$itemtype = 'Computer';
$p = [
    'addhidden' => [ // some hidden inputs added to the criteria form
        'hidden_input' => 'OK'
    ],
    'actionname' => 'preview', //change the submit button name
    'actionvalue' => __('Preview'), //change the submit button label
];
Search::showGenericSearch($itemtype, $p);
```


If you want to display only a list without the criteria form:

```
<?php
// display a list of users with entity = 'Root entity'
$itemtype = 'User';
$p = [
    'start'      => 0,          // start with first item (index 0)
    'is_deleted' => 0,          // item is not deleted
    'sort'       => 1,          // sort by name
    'order'      => 'DESC'      // sort direction
    'reset'      => 'reset',    // reset search flag
    'criteria'   => [
        [
            'field'      => 80,    // field index in search options
            'searchtype' => 'equals', // type of search
            'value'      => 0,    // value to search
        ],
    ],
];
Search::showList($itemtype, $p);
```

3.3.2 GET Parameters



Note: GLPI saves in `$_SESSION['glpisearch'][$itemtype]` the last set of parameters for the current itemtype for each search query. It is automatically restored on a new search if no reset, criteria or metacriteria is defined.

Here is the list of possible keys which could be passed to control the search engine. All are optional.

criteria

An multi-dimensional array of criterion to filter the search. Each criterion array must provide:

- link: one of *AND*, *OR*, *AND NOT* or *OR NOT* logical operators, optional for first element,
- field: id of the *searchoption*,
- searchtype: type of search, one of:
 - contains
 - equals
 - notequals
 - lessthan

- morethan
- under
- notunder
- value: the value to search

Note: In order to find the field id you want, you may take a look at the [getsearchoptions.php tool script](#).

metacriteria

Very similar to [criteria parameter](#) but permits to search in the [search options](#) of an itemtype linked to the current (the software of a computer, for example).

Not all itemtype can be linked, see the `getMetaItemtypeAvailable()` method of the Search class to know which ones could be.

The parameter need the same keys as criteria plus one additional:

- *itemtype*: second itemtype to link.

sort

id of the searchoption to sort by.

order

Either ASC for ending sorting or DESC for ending sorting.

start

An integer to indicate the start point of pagination (SQL OFFSET).

is_deleted

A boolean for display trash-bin.

reset

A boolean to reset saved search parameters, see note below.

3.3.3 Search options

Each itemtype can define a set of options to represent the columns which can be queried/displayed by the search engine. Each option is identified by an unique integer (we must avoid conflict).

Changed in version 9.2: Searchoptions array has been completely rewritten; mainly to catch duplicates and add a unit test to prevent future issues.

To permit the use of both old and new syntax; a new method has been created, `getSearchOptionsNew()`. Old syntax is still valid (but do not permit to catch duplicates).

The format has changed, but not the possible options and their values!

```
<?php
function getSearchOptionsNew() {
    $stab = [];

    $stab[] = [
        'id'           => 'common',
        'name'         => __('Characteristics')
    ];
}
```

(continues on next page)

(continued from previous page)

```

$tab[] = [
    'id'           => '1',
    'table'        => self::getTable(),
    'field'        => 'name',
    'name'         => __('Name'),
    'datatype'     => 'itemlink',
    'massiveaction' => false
];

...

return $tab;
}

```

Note: For reference, the old way to write the same search options was:

```

<?php
function getSearchOptions() {
    $tab = array();
    $tab['common'] = __('Characteristics');

    $tab[1]['table'] = self::getTable();
    $tab[1]['field'] = 'name';
    $tab[1]['name'] = __('Name');
    $tab[1]['datatype'] = 'itemlink';
    $tab[1]['massiveaction'] = false;

    ...

    return $tab;
}

```

Each option **must** define the following keys:

table

The SQL table where the `field` key can be found.

field

The SQL column to query.

name

A label used to display the *search option* in the search pages (like header for example).

Optionally, it can defined the following keys:

linkfield

Foreign key used to join to the current itemtype table. If not empty, standard massive action (update feature) for this *search option* will be impossible

searchtype

A string or an array containing forced search type:

- `equals` (may force use of field instead of id when adding `searchequalsonfield` option)

- contains

forcegroupby

A boolean to force group by on this *search option*

splititems

Use <hr> instead of
 to split grouped items

usehaving

Use HAVING SQL clause instead of WHERE in SQL query

massiveaction

Set to false to disable the massive actions for this *search option*.

nosort

Set to true to disable sorting with this *search option*.

nosearch

Set to true to disable searching in this *search option*.

nodisplay

Set to true to disable displaying this *search option*.

joinparams

Defines how the SQL join must be done. See [paragraph on joinparams](#) below.

additionalfields

An array for additional fields to add in the SELECT clause. For example: 'additionalfields' => ['id', 'content', 'status']

datatype

Define how the *search option* will be displayed and if a control need to be used for modification (ex: datepicker for date) and affect the *searchtype* dropdown. *optional parameters* are added to the base array of the *search option* to control more exactly the datatype.

See the [datatype paragraph](#) below.

Join parameters

To define join parameters, you can use one or more of the following:

beforejoin

Define which tables must be joined to access the field.

The array contains `table` key and may contain an additional `joinparams`. In case of nested `beforejoin`, we start the SQL join from the last dimension.

Example:

```
<?php
[
    'beforejoin' => [
        'table'      => 'mytable',
        'joinparams' => [
            'beforejoin' => [...]
        ]
    ]
]
```

jointype

Define the join type:

- empty for a standard jointype::

```
REFTABLE.`#linkfield#` = NEWTABLE.`id`
```

- child for a child table::

```
REFTABLE.`id` = NEWTABLE.`#linkfield#`
```

- itemtype_item for links using itemtype and items_id fields in new table::

```
REFTABLE.`id` = NEWTABLE.`items_id`  
AND NEWTABLE.`itemtype` = '#ref_table_itemtype#'
```

- itemtype_item_revert (since 9.2.1) for links using itemtype and items_id fields in ref table::

```
NEWTABLE.`id` = REFTABLE.`items_id`  
AND REFTABLE.`itemtype` = '#new_table_itemtype#'
```

- mainitemtype_mainitem same as itemtype_item but using mainitemtype and mainitems_id fields::

```
REFTABLE.`id` = NEWTABLE.`mainitems_id`  
AND NEWTABLE.`mainitemtype` = 'new table itemtype'
```

- itemtypeonly same as itemtype_item jointype but without linking id::

```
NEWTABLE.`itemtype` = '#new_table_itemtype#'
```

- item_item for table used to link two similar items: glpi_tickets_tickets for example: link fields are standardfk_1 and standardfk_2::

```
REFTABLE.`id` = NEWTABLE.`#fk_for_new_table#_1`  
OR REFTABLE.`id` = NEWTABLE.`#fk_for_new_table#_2`
```

- item_item_revert same as item_item and child jointypes::

```
NEWTABLE.`id` = REFTABLE.`#fk_for_new_table#_1`  
OR NEWTABLE.`id` = REFTABLE.`#fk_for_new_table#_2`
```

condition

Additional condition to add to the standard link.

Use NEWTABLE or REFTABLE tag to use the table names.

Changed in version 9.4.

An array of parameters used to build a *WHERE* clause from *GLPI querying facilities*. Was previously only a string.

nolink

Set to true to indicate the current join does not link to the previous join/from (nested joinparams)

Data types

Available datatypes for search are:

date

Available parameters (all optional):

- `searchunit`: one of [MySQL DATE_ADD unit](#), default to `MONTH`
- `maybefuture`: display datepicker with future date selection, defaults to `false`
- `emptylabel`: string to display in case of null value

datetime

Available parameters (all optional) are the same as `date`.

date_delay

Date with a delay in month (`end_warranty`, `end_date`).

Available parameters (all optional) are the same as `date` and:

- `datafields`: array of data fields that would be used.
 - `datafields[1]`: the date field,
 - `datafields[2]`: the delay field,
 - `datafields[2]`: ?
- `delay_unit`: one of [MySQL DATE_ADD unit](#), default to `MONTH`

timestamp

Use `Dropdown::showTimeStamp()` for modification

Available parameters (all optional):

- `withseconds`: boolean (`false` by default)

weblink

Any URL

email

Any email address

color

Use `Html::showColorField()` for modification

text

Use text area input for modification (optionally rich-text)

string

Simple, single-line text

ip

Any IP address

mac

Available parameters (all optional):

- `htmltext`: boolean, escape the value (`false` by default)

number

Use a `Dropdown::showNumber()` for modification (in case of equals searchtype). For contains searchtype, you can use < and > prefix in value.

Available parameters (all optional):

- **width**: html attribute passed to `Dropdown::showNumber()`
- **min**: minimum value (default 0)
- **max**: maximum value (default 100)
- **step**: step for select (default 1)
- **toadd**: array of values to add a the beginning of the dropdown

integer

Alias for **number**

count

Same as **number** but count the number of item in the table

decimal

Same as **number** but formatted with decimal

bool

Use `Dropdown::showYesNo()` for modification

itemlink

Create a link to the item

itemtypename

Use `Dropdown::showItemTypes()` for modification

Available parameters (all optional) to define available itemtypes:

- **itemtype_list**: one of `$CFG_GLPI["unicity_types"]`
- **types**: array containing available types

language

Use `Dropdown::showLanguages()` for modification

Available parameters (all optional):

- **display_emptychoice**: display an empty choice (-----)

right

Use `Profile::dropdownRights()` for modification

Available parameters (all optional):

- **nonone**: hide none choice ? (defaults to false)
- **noread**: hide read choice ? (defaults to false)
- **nowrite**: hide write choice ? (defaults to false)

dropdown

Use `Itemtype::dropdown()` for modification. Dropdown may have several additional parameters depending of dropdown type : **right** for user one for example

specific

If not any of the previous options matches the way you want to display your field, you can use this datatype. See [specific search options](#) paragraph for implementation.

Specific search options

You may want to control how to select and display your field in a searchoption. You need to set 'datatype' => 'specific' in your search option and declare these methods in your class:

getSpecificValueToDisplay

Define how to display the field in the list.

Parameters:

- **\$field**: column name, it matches the 'field' key of your searchoptions
- **\$values**: all the values of the current row (for select)
- **\$options**: will contains these keys:
 - **html**,
 - **searchopt**: the current full searchoption

getSpecificValueToSelect

Define how to display the field input in the criteria form and massive action.

Parameters:

- **\$field**: column name, it matches the 'field' key of your searchoptions
- **\$values**: the current criteria value passed in \$_GET parameters
- **\$name**: the html attribute name for the input to display
- **\$options**: this array may vary strongly in function of the searchoption or from the massiveaction or criteria display. Check the corresponding files:
 - [searchoptionvalue.php](#)
 - [massiveaction.class.php](#)

Simplified example extracted from CommonItilObject Class for `glpi_tickets.status` field:

```
<?php

function getSearchOptionsMain() {
    $tab = [];

    ...

    $tab[] = [
        'id'           => '12',
        'table'        => $this->getTable(),
        'field'        => 'status',
        'name'         => __('Status'),
        'searchtype'   => 'equals',
        'datatype'     => 'specific'
    ];
}
```

(continues on next page)

(continued from previous page)

```

...

return $tab;
}

static function getSpecificValueToDisplay($field, $values, array $options=array()) {

    if (!is_array($values)) {
        $values = array($field => $values);
    }
    switch ($field) {
        case 'status':
            return self::getStatus($values[$field]);

            ...

    }
    return parent::getSpecificValueToDisplay($field, $values, $options);
}

static function getSpecificValueToSelect($field, $name='', $values='', array
↪$options=array()) {

    if (!is_array($values)) {
        $values = array($field => $values);
    }
    $options['display'] = false;

    switch ($field) {
        case 'status' :
            $options['name'] = $name;
            $options['value'] = $values[$field];
            return self::dropdownStatus($options);

            ...

    }
    return parent::getSpecificValueToSelect($field, $name, $values, $options);
}

```

3.3.4 Default Select/Where/Join

The search class implements three methods which add some stuff to SQL queries before the searchoptions computation. For some itemtype, we need to filter the query or additional fields to it. For example, filtering the tickets you cannot view if you do not have the proper rights.

GLPI will automatically call predefined methods you can rely on from your plugin hook.php file.

addDefaultSelect

See `addDefaultSelect()` method documentation

And in the plugin hook.php file:

```
<?php
function plugin_mypluginname_addDefaultSelect($itemtype) {
    switch ($type) {
        case 'MyItemtype':
            return "`mytable`.`myfield` = 'myvalue' AS MYNAME, ";
    }
    return '';
}
```

addDefaultWhere

See `addDefaultWhere()` method documentation

And in the plugin hook.php file:

```
<?php
function plugin_mypluginname_addDefaultJoin($itemtype, $ref_table, &$already_link_
↪tables) {
    switch ($itemtype) {
        case 'MyItemtype':
            return Search::addLeftJoin(
                $itemtype,
                $ref_table,
                $already_link_tables,
                'newtable',
                'linkfield'
            );
    }
    return '';
}
```

addDefaultJoin

See `addDefaultJoin()`

And in the plugin hook.php file:

```
<?php
function plugin_mypluginname_addDefaultWhere($itemtype) {
    switch ($itemtype) {
        case 'MyItemtype':
            return "`mytable`.`myfield` = 'myvalue' ";
    }
    return '';
}
```

3.3.5 Bookmarks

The `glpi_bookmarks` table stores a list of search queries for the users and permit to retrieve them.

The query field contains an url query construct from *parameters* with `http_build_query` PHP function.

3.3.6 Display Preferences

The `glpi_displaypreferences` table stores the list of default columns which need to be displayed to a user for an itemtype.

A set of preferences can be *personal* or *global* (`users_id = 0`). If a user does not have any personal preferences for an itemtype, the search engine will use the global preferences.



3.4 Massive Actions



3.4.1 Goals

Add to itemtypes *search lists*:

- a checkbox before each item,
- a checkbox to select all items checkboxes,
- an *Actions* button to apply modifications to each selected items.

3.4.2 Update item's fields

The first option of the Actions button is Update. It permits to modify the fields content of the selected items.

The list of fields displayed in the sub list depends on the *Search options* of the current itemtype. By default, all *Search options* are automatically displayed in this list. To forbid this display for one field, you must define the key *massiveaction* to false in the *Search options* declaration, example:

```
<?php
$tab[] = [
    'id'           => '1',
    'table'        => self::getTable(),
    'field'        => 'name',
    'name'         => __('Name'),
    'datatype'     => 'itemlink',
    'massiveaction' => false // <- NO MASSIVE ACTION
];
```

3.4.3 Specific massive actions

After the Update entry, we can declare additional specific massive actions for our current itemtype.

First, we need declare in our class a *getSpecificMassiveActions* method containing our massive action definitions:

```
<?php
...

function getSpecificMassiveActions($checkitem=NULL) {
    $actions = parent::getSpecificMassiveActions($checkitem);

    // add a single massive action
    $class      = __CLASS__;
    $action_key = "myaction_key";
    $action_label = "My new massive action";
    $actions[$class.MassiveAction::CLASS_ACTION_SEPARATOR.$action_key] = $action_label;

    return $actions;
}
```

A single declaration is defined by these parts:

- a classname
- a separator: always `MassiveAction::CLASS_ACTION_SEPARATOR`
- a key
- and a label

We can have multiple actions for the same class, and we may target different class from our current object.

Next, to display the form of our definitions, we need to declare a *showMassiveActionsSubForm* method:

```

<?php
...

static function showMassiveActionsSubForm(MassiveAction $ma) {
    switch ($ma->getAction()) {
        case 'myaction_key':
            echo __("fill the input");
            echo Html::input('myinput');
            echo Html::submit(__('Do it'), array('name' => 'massiveaction'))."</span>";

            break;
    }

    return parent::showMassiveActionsSubForm($ma);
}

```

Finally, to process our definition, we need a `processMassiveActionsForOneItemtype` method:

```

<?php
...

static function processMassiveActionsForOneItemtype(MassiveAction $ma, CommonDBTM $item,
                                                    array $ids) {
    switch ($ma->getAction()) {
        case 'myaction_key':
            $input = $ma->getInput();

            foreach ($ids as $id) {
                if ($item->getFromDB($id)
                    && $item->doIt($input)) {
                    $ma->itemDone($item->getType(), $id, MassiveAction::ACTION_OK);
                } else {
                    $ma->itemDone($item->getType(), $id, MassiveAction::ACTION_KO);
                    $ma->addMessage(__("Something went wrong"));
                }
            }
            return;
    }

    parent::processMassiveActionsForOneItemtype($ma, $item, $ids);
}

```

Besides an instance of `MassiveAction` class `$ma`, we have also an instance of the current `itemtype` `$item` and the list of selected id ``\$ids`.

In this method, we could use some optional utility functions from the `MassiveAction` `$ma` object supplied in parameter :

- `itemDone`, indicates the result of the current `$id`, see constants of `MassiveAction` class. If we miss this call, the current `$id` will still be considered as OK.
- `addMessage`, a string to send to the user for explaining the result when processing the current `$id`



3.5 Rules Engine

GLPI provide a set of tools to implements a rule engine which take **criteria** in input and output actions. **criteria** and actions are defined by the user (and/or predefined at the GLPI installation).

Here is the list of base rules set provided in a staple GLPI:

- **ruleimportentity**: rules for assigning an item to an entity,
- **ruleimportcomputer**: rules for import and link computers,
- **rulemailcollector**: rules for assigning a ticket created through a mails receiver,
- **rulerright**: authorizations assignment rules,
- **rulesoftwarecategory**: rules for assigning a category to software,
- **ruleticket**: business rules for ticket.

Plugin could add their own set of rules.

3.5.1 Classes

A rules system is represented by these base classes:

- **Rule class**
Parent class for all Rule* classes. This class represents a single rule (matching a line in `glpi_rules` table) and include test, process, display for an instance.
- **RuleCollection class**
Parent class for all Rule*Collection classes.
This class represents the whole collection of rules for a `sub_type` (matching all line in `glpi_rules` table for this `sub_type`) and includes some method to process, duplicate, test and display the full collection.
- **RuleCriteria class**
This class permits to manipulate a single criteria (matching a line in `glpi_rulecriterias` table) and include methods to display and match input values.
- **RuleAction class**
This class permits to manipulate a single action (matching a line in `glpi_ruleactions` table) and include methods to display and process output values.

And for each `sub_type` of rule:

- **RuleSubtype class**
Define the specificity of the `sub_type` rule like list of criteria and actions or how to display specific parts.
- **RuleSubtypeCollection class**
Define the specificity of the `sub_type` rule collection like the preparation of input and the tests results.

3.5.2 Database Model

Here is the list of important tables / fields for rules:

- **glpi_rules:**

All rules for all **sub_types** are inserted here.

- **sub_type:** the type of the rule (ruleticket, ruleright, etc),
- **ranking:** the order of execution in the collection,
- **match:** define the link between the rule's criteria. Can be AND or OR,
- **uuid:** unique id for the rule, useful for import/export in xml,
- **condition:** addition condition for the **sub_type** (only used by ruleticket for defining the trigger of the collection on add and/or update of a ticket).

- **glpi_rulecriterias:**

Store all criteria for all rules.

- **rules_id:** the foreign key for **glpi_rules**,
- **criteria:** one of the key defined in the `RuleSubtype::getCriteria()` method,
- **condition:** an integer matching the constant set in `Rule` class constants,
- **pattern:** the direct value or regex to compare to the criteria.

- **glpi_ruleactions:**

Store all actions for all rules.

- **rules_id:** the foreign key for **glpi_rules**,
- **action_type:** the type of action to apply on the input. See `RuleAction::getActions()`,
- **field:** the field to alter by the current action. See keys definition in `RuleSubtype::getActions()`,
- **value:** the value to apply in the field.

3.5.3 Add a new Rule class

Here is the minimal setup to have a working set. You need to add the following classes for describing you new **sub_type**.

- `src/RuleMytype.php`

```
<?php

class RuleMytype extends Rule {

    // optional right to apply to this rule type (default: 'config', see Rights_
↪management.
    static $rightname = 'rule_mytype';

    // define a label to display in interface titles
    function getTitle() {
        return __('My rule type name');
```

(continues on next page)

(continued from previous page)

```

}

// return an array of criteria
function getCriteria() {
    $criteria = [
        '_users_id_requester' => [
            'field'    => 'name',
            'name'     => __('Requester'),
            'table'    => 'glpi_users',
            'type'     => 'dropdown',
        ],
        'GROUPS'          => [
            'table'     => 'glpi_groups',
            'field'     => 'completename',
            'name'      => sprintf(__('%1$s: %2$s'), __('User'),
                                __('Group'));
            'linkfield' => '',
            'type'      => 'dropdown',
            'virtual'   => true,
            'id'        => 'groups',
        ],
        ...
    ];

    $criteria['GROUPS']['table']          = 'glpi_groups';
    $criteria['GROUPS']['field']          = 'completename';
    $criteria['GROUPS']['name']           = sprintf(__('%1$s: %2$s'), __('
    ↪ User'),
                                                __('Group'));

    $criteria['GROUPS']['linkfield']      = '';
    $criteria['GROUPS']['type']           = 'dropdown';
    $criteria['GROUPS']['virtual']        = true;
    $criteria['GROUPS']['id']             = 'groups';

    return $criteria;
}

// return an array of actions
function getActions() {
    $actions = [
        'entities_id' => [
            'name' => __('Entity'),
            'type' => 'dropdown',
            'table' => 'glpi_entities',
        ],
        ...
    ];
}

```

(continues on next page)

(continued from previous page)

```

        return $actions;
    }
}

```

- src/RuleMytypeCollection.php

```

<?php

class RuleMytypeCollection extends RuleCollection {
    // a rule collection can process all rules for the input or stop
    //after a single match with its criteria (default false)
    public $stop_on_first_match = true;

    // optional right to apply to this rule type (default: 'config'),
    //see Rights management.
    static $rightname = 'rule_mytype';

    // menu key to use with Html::header in front page.
    public $menu_option = 'myruletype';

    // define a label to display in interface titles
    function getTitle() {
        return __('My rule type name');
    }

    // if we need to change the input of the object before passing
    //it to the criteria.
    // Example if the input couldn't directly contains a criteria
    //and we need to compute it before (GROUP)
    function prepareInputDataForProcess($input, $params) {
        $input['_users_id_requester'] = $params['_users_id_requester'];
        $fields = $this->getFieldsToLookFor();

        //Add all user's groups
        if (in_array('groups', $fields)) {
            foreach (Group_User::getUserGroups($input['_users_id_requester']) as $group)
                $input['GROUPS'][] = $group['id'];
        }

        ...

        return $input;
    }
}

```

You need to also add the following php files for list and form:

- front/rulemytype.php

```
<?php
include ('../inc/includes.php');
$rulecollection = new RuleMytypeCollection($_SESSION['glpiactive_entity']);
include (GLPI_ROOT . "/front/rule.common.php");
```

- front/rulemytype.form.php

```
<?php
include ('../inc/includes.php');
$rulecollection = new RuleMytypeCollection($_SESSION['glpiactive_entity']);
include (GLPI_ROOT . "/front/rule.common.form.php");
```

And add the rulecollection in \$CFG_GLPI (Only for **Core** rules):

- inc/define.php

```
<?php

...

$CFG_GLPI["rulecollections_types"] = [
    'RuleImportEntityCollection',
    'RuleImportComputerCollection',
    'RuleMailCollectorCollection',
    'RuleRightCollection',
    'RuleSoftwareCategoryCollection',
    'RuleTicketCollection',
    'RuleMytypeCollection' // <-- My type is added here
];
```

Plugin instead must declare it in *their init function*:

- plugin/myplugin/setup.php

```
<?php
function plugin_init_myplugin() {
    ...

    $Plugin->registerClass(
        'PluginMypluginRuleMytypeCollection',
        ['rulecollections_types' => true]
    );

    ...
}
```

3.5.4 Apply a rule collection

To call your rules collection and alter the data:

```
<?php
...

$rules = new PluginMypluginRuleMytypeCollection();

// data send by a form (which will be compared to criteria)
$input = [...];
// usually = $input, but it could differ if you want to avoid comparison of
//some fields with the criteria.
$output = [...];
// array passed to the prepareInputDataForProcess function of the collection
//class (if you need to add conditions)
$params = [];

$output = $rules->processAllRules(
    $input,
    $output,
    $params
);
```

3.5.5 Dictionaries

They inherits Rule* classes but have some specificities.

A dictionary aims to modify on the fly data coming from an external source (CSV file, inventory tools, etc.). It applies on an itemtype, as defined in the sub_type field of the glpi_rules table.

As the classic rules aim to apply additional and multiple data to input, dictionaries generally used to alter a single field (relative to the their sub_type). Ex, RuleDictionaryComputerModel alters model field of glpi_computers.

Some exceptions exists and provide multiple actions (Ex: RuleDictionarySoftware).

As they are shown in a separate menu, you should define they in a separate \$CFG_GLPI entry in inc/define.php:

```
<?php
...

$CFG_GLPI["dictionary_types"] = array('ComputerModel', 'ComputerType', 'Manufacturer',
    'MonitorModel', 'MonitorType',
    'NetworkEquipmentModel', 'NetworkEquipmentType',
    'OperatingSystem', 'OperatingSystemServicePack',
    'OperatingSystemVersion', 'PeripheralModel',
    'PeripheralType', 'PhoneModel', 'PhoneType',
    'Printer', 'PrinterModel', 'PrinterType',
    'Software', 'OperatingSystemArchitecture',
    'RuleMytypeCollection' // <-- My type is added.
    ↪here
);
```



3.6 Translations

Main GLPI language is british english (en_GB). All string in the source code must be in english, and marked as translatable, using some convenient functions.

Since 0.84; GLPI uses [gettext](#) for localization; and [Transifex](#) is used for translations. If you want to help translating GLPI, please register on transifex and join our [translation mailing list](#)

What the system is capable to do:

- replace variables (on LTR and RTL languages),
- manage plural forms,
- add context information,
- ...

Here is the workflow used for translations:

1. Developers add string in the source code,
2. String are extracted to POT file,
3. POT file is sent to Transifex,
4. Translators translate,
5. Developers pull new translations from Transifex,
6. MO files used by GLPI are generated.

3.6.1 PHP Functions

There are several standard functions you will have to use in order to get translations. Remember the translation domain will be *glpi* if not defined; so, for plugins specific translations, do not forget to set it!

Note: All translations functions take a `$domain` as argument; it defaults to `glpi` and must be changed when you are working on a plugin.

Simple translation

When you have a “simple” string to translate, you may use several functions, depending on the particular use case:

- `__($str, $domain='glpi')` (what you will probably use the most frequently): just translate a string,
- `_x($ctx, $str, $domain='glpi')`: same as `__()` but provide an extra context,
- `__s($str, $domain='glpi')`: same as `__()` but escape HTML entities,
- `_sx($ctx, $str, $domain='glpi')`: same as `__()` but provide an extra context and escape HTML entities,

Handle plural forms

When you have a string to translate, but which rely on a count or something. You may as well use several functions, depending on the particular use case:

- `_n($sing, $plural, $nb, $domain='glpi')` (what you will probably use the most frequently): give a string for singular form, another for plural form, and set current “count”,
- `_sn($str, $domain='glpi')`: same as `_n()` but escape HTML entities,
- `_nx($ctx, $str, $domain='glpi')`: same as `_n()` but provide an extra context,

Handle variables

You may want to replace some parts of translations; for some reason. Let's say you would like to display current page on a total number of pages; you will use the `sprintf` method. This will allow you to make replacements; but without relying on arguments positions. For example:

```
<?php
$pages = 20; //total number of pages
$current = 2; //current page
$string = sprintf(
    __('Page %1$s on %2$s'),
    $pages,
    $total
);
echo $string; //will display: "Page 2 on 20"
```

In the above example, %1\$s will always be replaced by 2; even if places has been changed in some translations.

Warning: You may sometimes see the use of `printf()` which is an equivalent that directly output (echo) the result. This should be avoided!

3.6.2 Javascript Functions

New in version 9.5.0.

Translation functions `__()`, `_x()`, `_n()`, `_nx()` are also available in javascript in browser context. They have same signatures as PHP functions.

```
alert(__('Test successful'));
```



3.7 Right Management

3.7.1 Goals

Provide a way for administrator to segment usages into profiles of users.

3.7.2 Profiles

The Profile (corresponding to `glpi_profiles` table) stores each set of rights.

A profile has a set of base fields independent of sub rights and, so, could:

- be defined as default for new users (`is_default` field).
- force the ticket creation form at the login (`create_ticket_on_login` field).
- define the interface used (`interface` field):
 - helpdesk (self-service users)
 - central (technician view)

3.7.3 Rights definition

They are defined by the `ProfileRight` class (corresponding to `glpi_profilerights` table)

Each consists of:

- a profile foreign key (`profiles_id` field)
- a key (`name` field)
- a value (`right` field)

The keys match the static property `$rightname` in the GLPI itemtypes. Ex: In `Computer` class, we have a static `$rightname = 'computer'`;

Value is a numeric sum of integer constants.

Values of standard rights can be found in `inc/define.php`:

```
<?php
...

define("READ", 1);
define("UPDATE", 2);
define("CREATE", 4);
define("DELETE", 8);
define("PURGE", 16);
define("ALLSTANDARDRIGHT", 31);
define("READNOTE", 32);
define("UPDATENOTE", 64);
define("UNLOCK", 128);
```

So, for example, to have the right to `READ` and `UPDATE` an itemtype, we'll have a `right` value of 3.

As defined in this above block, we have a computation of all standards right = 31:

```

READ (1)
\+ UPDATE (2)
\+ CREATE (4)
\+ DELETE (8)
\+ PURGE (16)
= 31

```

If you need to extend the possible values of rights, you need to declare these parts into your itemtype, simplified example from Ticket class:

```

<?php

class Ticket extends CommonITILObject {

    ...

    const READALL          = 1024;
    const READGROUP        = 2048;

    ...

    function getRights($interface = 'central') {
        $values = parent::getRights();

        $values[self::READGROUP] = array('short' => __('See group ticket'),
                                          'long'  => __('See tickets created by my groups
→'));

        $values[self::READASSIGN] = array('short' => __('See assigned'),
                                          'long'  => __('See assigned tickets'));

        return $values;
    }

    ...

```

The new rights need to be checked by your own functions, see [check rights](#)

3.7.4 Check rights

Each itemtype class which inherits from CommonDBTM will benefit from standard right checks. See the following methods:

- canView
- canUpdate
- canCreate
- canDelete
- canPurge

If you need to test a specific rightname against a possible right, here is how to do:

```
<?php

if (Session::haveRight(self::$rightname, CREATE)) {
    // OK
}

// we can also test a set multiple rights with AND operator
if (Session::haveRightsAnd(self::$rightname, [CREATE, READ])) {
    // OK
}

// also with OR operator
if (Session::haveRightsOr(self::$rightname, [CREATE, READ])) {
    // OK
}

// check a specific right (not your class one)
if (Session::haveRight('ticket', CREATE)) {
    // OK
}
```

See methods definition:

- haveRight
- haveRightsAnd
- haveRightsOr

All above functions return a boolean. If we want a graceful die of your pages, we have equivalent function but with a check prefix instead have:

- checkRight
- checkRightsAnd
- checkRightsOr

If you need to check a right directly in a SQL query, use bitwise `&` and `|` operators, ex for users:

```
<?php

$query = "SELECT `glpi_profiles_users`.`users_id`
FROM `glpi_profiles_users`
INNER JOIN `glpi_profiles`
    ON (`glpi_profiles_users`.`profiles_id` = `glpi_profiles`.`id`)
INNER JOIN `glpi_profilerrights`
    ON (`glpi_profilerrights`.`profiles_id` = `glpi_profiles`.`id`)
WHERE `glpi_profilerrights`.`name` = 'ticket'
    AND `glpi_profilerrights`.`rights` & ". (READ | CREATE);
$result = $DB->query($query);
```

In this snippet, the `READ | CREATE` do a bitwise operation to get the sum of these rights and the `&` SQL operator do a logical comparison with the current value in the DB.

3.7.5 CommonDBRelation and CommonDBChild specificities

These classes permits to manage the relation between items and so have properties to propagate rights from their parents.

```
<?php

abstract class CommonDBChild extends CommonDBConnexity {
    static public $checkParentRights = self::HAVE_SAME_RIGHT_ON_ITEM;

    ...
}

abstract class CommonDBRelation extends CommonDBConnexity {
    static public $checkItem_1_Rights = self::HAVE_SAME_RIGHT_ON_ITEM;
    static public $checkItem_2_Rights = self::HAVE_SAME_RIGHT_ON_ITEM;

    ...
}
```

possible values for these properties are:

- DONT_CHECK_ITEM_RIGHTS: don't check the parent, we always have all rights regardless of parent's rights.
- HAVE_VIEW_RIGHT_ON_ITEM: we have all rights (CREATE, UPDATE), if we can view the parent.
- HAVE_SAME_RIGHT_ON_ITEM: we have the same rights as the parent class.



3.8 Automatic actions

3.8.1 Goals

Provide a scheduler for background tasks used by GLPI and its plugins.

3.8.2 Implementation overview

The entry point of automatic actions is the file `front/cron.php`. On each execution, it executes a limited number of automatic actions.

There are two ways to wake up the scheduler :

- when a user browses in GLPI (the internal mode)
- when the operating system's scheduler calls `front/cron.php` (the external mode)

When GLPI generates an HTML page for a browser, it adds an invisible image generated by `front/cron.php`. This way, the automatic action runs in a separate process and does not impact the user.

The automatic actions are defined by the `CronTask` class. GLPI defines a lot of them for its own needs. They are created in the installation or upgrade process.

3.8.3 Implementation

Automatic actions could be related to an itemtype and the implementation is defined in its class or haven't any itemtype relation and are implemented directly into CronTask class.

When GLPI shows a list of automatic actions, it shows a short description for each item. The description is gathered in the static method `cronInfo()` of the itemtype.

Note: An itemtype may contain several automatic actions.

Example of implementation from the QueuedNotification:

```
<?php
class QueuedNotification extends CommonDBTM {

    // ...

    /**
     * Give cron information
     *
     * @param $name : automatic action's name
     *
     * @return array of information
     */
    static function cronInfo($name) {

        switch ($name) {
            case 'queuednotification' :
                return array('description' => __('Send mails in queue'),
                             'parameter'   => __('Maximum emails to send at once'));
        }
        return [];
    }

    /**
     * Cron action on notification queue: send notifications in queue
     *
     * @param CommonDBTM $task for log (default NULL)
     *
     * @return integer either 0 or 1
     */
    static function cronQueuedNotification($task=NULL) {
        global $DB, $CFG_GLPI;

        if (!$CFG_GLPI["notifications_mailing"]) {
            return 0;
        }
        $cron_status = 0;

        // Send mail at least 1 minute after adding in queue to be sure that process on it
        ↪ is finished
        $send_time = date("Y-m-d H:i:s", strtotime("+1 minutes"));
    }
}
```

(continues on next page)

(continued from previous page)

```

$mail = new self();
$pendings = self::getPendings(
    $send_time,
    $task->fields['param']
);

foreach ($pendings as $mode => $data) {
    $eventclass = 'NotificationEvent' . ucfirst($mode);
    $conf = Notification_NotificationTemplate::getMode($mode);
    if ($conf['from'] != 'core') {
        $eventclass = 'Plugin' . ucfirst($conf['from']) . $eventclass;
    }

    $result = $eventclass::send($data);
    if ($result !== false && count($result)) {
        $cron_status = 1;
        if (!is_null($task)) {
            $task->addVolume($result);
        }
    }
}

return $cron_status;
}

// ...
}

```

If the argument `$task` is a `CronTask` object, the method must increment the quantity of actions done. In this example, each notification type reports the quantity of notification processed and is added to the task's volume.

3.8.4 Register an automatic actions

Automatic actions are defined in the empty schema located in `install/mysql/`. Use the existing sql queries creating rows in the table `glpi_crontasks` as template for a new automatic action.

To handle upgrade from a previous version, the new automatic actions must be added in the appropriate update file `install/update_xx_to_yy.php`.

```

<?php
// Register an automatic action
CronTask::register('QueuedNotification', 'QueuedNotification', MINUTE_TIMESTAMP,
    array(
        'comment' => '',
        'mode' => CronTask::MODE_EXTERNAL
    ));

```

The `register` method takes four arguments:

- `itemtype`: a string containing an itemtype name containing the automatic action implementation
- `name`: a string containing the name of the automatic action

- `frequency` the period of time between two executions in seconds (see `inc/define.php` for convenient constants)
- `options` an array of options

Note: The name of an automatic action is actually the method's name without the prefix `cron`. In the example, the method `cronQueuedNotification` implements the automatic action named `QueuedNotification`.



3.9 Tools

Different tools are available on the `tools` folder; here is a non exhaustive list of provided features.

3.9.1 locale/

The locale directory contains several scripts used to maintain *translations* along with Transifex services:

- `extract_template.sh` is used to extract translated string to the POT file (before sending it to Transifex),
- `locale/update_mo.pl` compiles MO files from PO file (after they've been updated from transifex).

3.9.2 genapidoc.sh

Generate GLPI phpdoc using `apigen`. `apigen` command must be available in your path.

Generated documentation will be available in the `api` directory.

3.9.3 convert_search_options.php

Search options have changed in GLPI 9.2 (see [PR #1396](#)). This script is a helper to convert existing search options to new way.

Note: The script output can probably **not be used as is**; but it would probably help you a lot!

3.9.4 make_release.sh

Builds GLPI release tarball:

- install and cleanup third party libraries,
- remove files and directories that should not be part of tarball,
- minify CSS and Javascript files,
- ...

3.9.5 modify_headers.pl

Update copyright header based on the contents of the HEADER file.

3.9.6 getsearchoptions.php

This script is designed to be called from a browser, or from the command line. It will display existing search options for an item specified with the `type` argument.

For example, open `http://localhost/glpi/tools/getsearchoptions.php?type=Computer`, and you will see search options for *Computer* type.

On command line, you can get the exact same result entering:

```
$ php tools/getsearchoptions.php --type=Computer
```

3.9.7 generate_bigdump.php

This script is designed to generate many data in your GLPI instance. It relies on the `generate_bigdump.function.php` file.

3.9.8 Not yet documented...

Note: Following scripts are not yet documented... Feel free to open a pull request to add them!

- `autoupdatelocales.sh`: Probably obsolete
- `check_dict.pl`
- `check_functions.pl`
- `checkforms.php`: Check forms opened / closed
- `checkfunction.php`: Check for obsolete function usage
- `cleanhistory.php`: Purge history with some criteria
- `diff_plugin_locale.php`: Probably obsolete
- `find_twin_in_dict.sh`: Check duplicates key in language template
- `findtableswithoutclass.php`
- `fix_utf8_bomfiles.sh`
- `fk_generate.php`
- `genphpcov.sh`
- `glpiuser.php`
- `ldap-glpi.ldif`: An LDAP export
- `ldap-schema.txt`: An LDAP export
- `ldapsync.php`
- `notincludedlanguages.php`: Get all po files not used in GLPI

- test_langfiles.php
- testmail.php
- testunit.php
- update_registered_ids.php: Purge history with some criteria

3.9.9 Out of date

Warning: Those tools are outdated, and kept for reference, or need some work to be working again. Use them at your own risks, or do not use them at all :)

phpunit/

This directory contains a set of unit tests that have not really been integrated in the project. Since, some unit tests have been rewritten, but not everything has been ported :/

php.vim

A vimfile for autocompletion and highlighting in VIM. This one is very outdated; it should be replaced with a most recent version, or being removed.



3.10 Extra

The extra config/local_define.php file will be loaded if present. It permit you to change some GLPI framework configurations.

3.10.1 Change logging level

Logging level is declared with the GLPI_LOG_LVL constant; and rely on [available Monolog levels](#). The default log level will change if debug mode is enabled on GUI or not. To change logging level to ERROR, add the following to your local_define.php file:

```
<?php
define('GLPI_LOG_LVL', \Monolog\Logger::ERROR);
```

Note: Once you've declared a logging level, it will **always be used**. It will no longer take care of the debug mode.

3.10.2 Override mailing recipient

In some cases, during development, you may want to test notifications that can be sent. Problem is you will have to make sure you are not going to send fake email to your real users if you rely on a production database copy for example.

You can define a unique email recipient for all emails that will be sent from GLPI. Original recipient address will be added as part of the message (for you to know who was originally targeted). To get all sent emails delivered on the *you@host.org* email address, use the `GLPI_FORCE_MAIL` in the `local_define.php` file:

```
<?php
define('GLPI_FORCE_MAIL', 'you@host.org');
```

3.10.3 Disabling CSRF checks

Warning: Use it with cautions!

While disabling CSRF checks may be really interesting during debugging, keep in mind that enabling it again (which is the default) may cause issues you cannot see before.

CSRF checks will prevent for example a same form to be sent twice. While this is the expected behavior for the application, this may be a pain during development or debugging. You can therefore use the `GLPI_USE_CSRF_CHECK` constant in the `local_define.php` file:

```
<?php
define('GLPI_USE_CSRF_CHECK', 0);
```



CHECKLISTS

Some really useful checklists, for development, releases, and so on!

4.1 Review process

Here is the process you must follow when you are reviewing a PR.

1. Make sure the destination branch is the correct one:
 - *master* for new features,
 - *xx/bugfixes* for bug fixes
2. Check if unit tests are not failing,
3. Check if coding standards checks are not failing,
4. Review the code itself. It must follow *GLPI's coding standards*,
5. Using the Github review process, approve, request changes or just comment the PR,
 - If some new methods are added, or if the request made important changes in the code, you should ask the developer to write some more unit tests
6. A PR can be merged if two developers approved it, or if one developer approved it more than one day ago,
7. A bugfix PR that has been merged into the *xx/bugfixes* branch must be reported on the *master* branch. If the *master* already contains many changes, you may have to change some code before doing this. If changes are consequent, maybe should you open a new PR against the *master* branch for it,
8. Say thanks to the contributor :-)



4.2 Prepare next major release

Once a major release has been finished, it's time to think about the next one!

You'll have to remember a few steps in order to get that working well:

- bump version in `config/define.php`
- create SQL empty script (copying last one) in `install/mysql/glpi-{version}-empty.sql`
- change empty SQL file calls in `inc/toolbox.class.php` (look for the `$DB->runFile` call)
- create a PHP migration script copying provided template `install/update_xx_xy.tpl.php`

- change its main comment to reflect reality
- change method name
- change version in `displayTitle` and `setVersion` calls
- add the new case in `install/update.php` and `tools/cliupdate.php`; that will include your new PHP migration script and then call the function defined in it
- change the `include` and the function called in the `--force` option part of the `tools/cliupdate.php` script

That's all, folks!



PLUGINS

GLPI provides facilities to develop plugins, and there are many [plugins that have been already published](#).

Note: Plugins are designed to add features to GLPI core.

This is a sub-directory in the `plugins` of GLPI; that would contains all related files.

Generally speaking, there is really a few things you have to do in order to get a plugin working; many considerations are up to you. Anyways, this guide will provide you some guidelines to get a plugins repository as consistent as possible :)

If you want to see more advanced examples of what it is possible to do with plugins, you can take a look at the [example plugin source code](#).










5.1 Guidelines

5.1.1 Directories structure

Real structure will depend of what your plugin propose. See [requirements](#) to find out what is needed. You may also want to [take a look at GLPI File Hierarchy Standard](#).

Warning: The main directory name of your plugin may contain only alphanumeric characters (no - or _ or accented characters or else).

The plugin directory structure should look like the following:

-  `MyPlugin`
 -  `front`
 - *  ...
 -  `inc`
 - *  ...
 -  `locale`
 - *  ...
 -  `tools`
 - *  ...

-  *README.md*
-  *LICENSE*
-  *setup.php*
-  *hook.php*
-  *MyPlugin.xml*
-  *MyPlugin.png*
-  ...
-  ...

- *front* will host all PHP files directly used to display something to the user,
- *inc* will host all classes,
- if you internationalize your plugin, localization files will be found under the *locale* directory,
- if you need any scripting tool (like something to extract or update your translatable strings), you can put them in the *tools* directory
- a *README.md* file describing the plugin features, how to install it, and so on,
- a *LICENSE* file containing the license,
- *MyPlugin.xml* and *MyPlugin.png* can be used to reference your plugin on the [plugins directory website](#),
- the required *setup.php* and *hook.php* files.

Where to write files?

Warning: Plugins my never ask user to give them write access on their own directory!

The GLPI installation already ask for administrator to get write access on its `files` directory; just use `GLPI_PLUGIN_DOC_DIR/{plugin_name}` (that would resolve to `glpi_dir/files/_plugins/{plugin_name}` in default basic installations).

Make sure to create the plugin directory at install time, and to remove it on uninstall.

5.1.2 Versionning

We recommend you to use [semantic versionning](#) for you plugins. You may find existing plugins that have adopted another logic; some have reasons, others don't... Well, it is up to you finally :-)

Whatever the versioning logic you adopt, you'll have to be consistent, it is not easy to change it without breaking things, once you've released something.

5.1.3 ChangeLog

Many projects make releases without providing any changelog file. It is not simple for any end user (whether a developer or not) to read a repository log or a list of tickets to know what have changed between two releases.

Keep in mind it could help users to know what have been changed. To achieve this, take a look at [Keep an ChangeLog](#), it will explain you some basics and give you guidelines to maintain sug a thing.

5.1.4 Third party libraries

Just like GLPI, you should use the *composer tool to manage third party libraries* for your plugin.



5.2 Requirements

- plugin will be installed by creating a directory in the `plugins` directory of the GLPI instance,
- plugin directory name should never change,
- each plugin **must** at least provides *setup.php* and *hook.php* files,
- if your plugin requires a newer PHP version than GLPI one, or extensions that are not mandatory in core; it is up to you to check that in the install process.

5.2.1 setup.php

The plugin's *setup.php* file will be automatically loaded from GLPI's core in order to get its version, to check pre-requisites, etc.

This is a good practice, thus not mandatory, to define a constant name `{PLUGINNAME}_VERSION` in this file.

This is a minimalist example, for a plugin named *myexample* (functions names will contain plugin name):

```
<?php

define('MYEXAMPLE_VERSION', '1.2.10');

/**
 * Init the hooks of the plugins - Needed
 *
 * @return void
 */
function plugin_init_myexample() {
    global $PLUGIN_HOOKS;

    //required!
    $PLUGIN_HOOKS['csrf_compliant']['myexample'] = true;

    //some code here, like call to Plugin::registerClass(), populating PLUGIN_HOOKS, ...
}

/**
```

(continues on next page)

(continued from previous page)

```

* Get the name and the version of the plugin - Needed
*
* @return array
*/
function plugin_version_myexample() {
    return [
        'name'           => 'Plugin name that will be displayed',
        'version'        => MYEXAMPLE_VERSION,
        'author'         => 'John Doe and <a href="http://foobar.com">Foo Bar</a>',
        'license'        => 'GLPv3',
        'homepage'       => 'http://perdu.com',
        'requirements'   => [
            'glpi' => [
                'min' => '9.1'
            ]
        ]
    ];
}

/**
 * Optional : check prerequisites before install : may print errors or add to message_
↳ after redirect
 *
 * @return boolean
 */
function plugin_myexample_check_prerequisites() {
    //do what the checks you want
    return true;
}

/**
 * Check configuration process for plugin : need to return true if succeeded
 * Can display a message only if failure and $verbose is true
 *
 * @param boolean $verbose Enable verbosity. Default to false
 *
 * @return boolean
 */
function plugin_myexample_check_config($verbose = false) {
    if (true) { // Your configuration check
        return true;
    }

    if ($verbose) {
        echo "Installed, but not configured";
    }
    return false;
}

/**
 * Optional: defines plugin options.
 *

```

(continues on next page)

(continued from previous page)

```

* @return array
*/
function plugin_myexample_options() {
    return [
        Plugin::OPTION_AUTOINSTALL_DISABLED => true,
    ];
}

```

Plugin information provided in `plugin_version_myexample` method will be displayed in the GLPI plugins user interface.

Requirements checking

Since GLPI 9.2; it is possible to provide some requirement information along with the information array. Those information are not mandatory, but we encourage you to migrate :)

Warning: Even if this has been deprecated for a while, many plugins continue to provide a `minGlpiVersion` entry in the information array. If this value is set; it will be automatically used as minimal GLPI version.

In order to set your requirements, add a `requirements` entry in the `plugin_version_myexample` information array. Let's say your plugin is compatible with a version of GLPI comprised between 0.90 and 9.2; with a minimal version of PHP set to 7.0. The method would look like:

```

<?php

function plugin_version_myexample() {
    return [
        'name'           => 'Plugin name that will be displayed',
        'version'         => MYEXAMPLE_VERSION,
        'author'          => 'John Doe and <a href="http://foobar.com">Foo Bar</a>',
        'license'          => 'GLPv3',
        'homepage'         => 'http://perdu.com',
        'requirements'    => [
            'glpi'         => [
                'min'       => '0.90',
                'max'       => '9.2'
            ],
            'php'          => [
                'min'       => '7.0'
            ]
        ]
    ];
}

```

`requirements` array may take the following values:

- `glpi`
 - `min`: minimal GLPI version required,
 - `max`: maximal supported GLPI version,
 - `dev`: whether the plugin is supported on development versions (*9.2-dev* for example),

- **params**: an array of GLPI parameters names that must be set (not empty, not null, not false),
- **plugins**: an array of plugins name your plugin depends on (must be installed and active).
- **php**
 - **min**: minimal PHP version required,
 - **max**: maximal PHP version supported (discouraged),
 - **params**: an array of parameters name that must be set (retrieved from `ini_get()`),
 - **exts**: array of used extensions (see below).

PHP extensions checks rely on core capabilities. You have to provide a multidimensional array with extension name as key. For each of those entries; you can define if the extension is required or not, and optionally a class or a function to check.

The following example is from the core:

```
<?php
$extensions = [
    'mysqli' => [
        'required' => true
    ],
    'fileinfo' => [
        'required' => true,
        'class' => 'finfo'
    ],
    'json' => [
        'required' => true,
        'function' => 'json_encode'
    ],
    'imap' => [
        'required' => false
    ]
];
```

- the `mysqli` extension is mandatory; `extension_loaded()` function will be used for check;
- the `fileinfo` extension is mandatory; `class_exists()` function will be used for check;
- the `json` extension is mandatory; `function_exists()` function will be used for check;
- the `imap` extension is not mandatory.

Note: Optional extensions are not yet handled in the checks function; but will probably be in the future. You can add them to the configuration right now :)

Without using automatic requirements; it's up to you to check with something like the following in the `plugin_myexample_check_prerequisites`:

Warning: Automatic requirements and manual checks are not exclusive. Both will be played! If you want to use automatic requirements with GLPI ≥ 9.2 and still provide manual checks for older versions; be careful not to indicate different versions.


```

<?php
// Version check
if (version_compare(GLPI_VERSION, '9.1', 'lt') || version_compare(GLPI_VERSION, '9.2',
↪ 'ge')) {
    if (method_exists('Plugin', 'messageIncompatible')) {
        //since GLPI 9.2
        Plugin::messageIncompatible('core', 9.1, 9.2);
    } else {
        echo "This plugin requires GLPI >= 9.1 and < 9.2";
    }
    return false;
}
}

```

Note: Since GLPI 9.2, you can rely on `Plugin::messageIncompatible()` to display internationalized messages when GLPI or PHP versions are not met.

On the same model, you can use `Plugin::messageMissingRequirement()` to display internationalized message if any extension, plugin or GLPI parameter is missing.

Plugin options

Since GLPI 10.0, it is possible to define some plugin options.

autoinstall_disabled

New in version 10.0.0.

Disable automatic call of plugin install hook function. For instance, when the plugin will be downloaded from GLPI marketplace, *plugin_myexample_install* will not be executed automatically. Administrator will have to use the “Install” or “Update” button to trigger this hook.

5.2.2 hook.php

This file will contains hooks that GLPI may call under some user actions. Refer to core documentation to know more about available hooks.

For instance, a plugin need both an install and an uninstall hook calls. Here is the minimal file:

```

<?php
/**
 * Install hook
 *
 * @return boolean
 */
function plugin_myexample_install() {
    //do some stuff like instantiating databases, default values, ...
    return true;
}

/**
 * Uninstall hook
 *

```

(continues on next page)

(continued from previous page)

```
* @return boolean
*/
function plugin_myexample_uninstall() {
    //to some stuff, like removing tables, generated files, ...
    return true;
}
```

5.2.3 Coding standards

You must respect GLPI's *global coding standards*.

In order to check for coding standards compliance, you can add the *glpi-project/coding-standard* to your composer file, using:

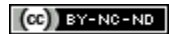
```
$ composer require --dev glpi-project/coding-standard
```

This will install the latest version of the coding-standard used in GLPI core. If you want to use an older version of the checks (for example if you have a huge amount of work to fix!), you can specify a version in the above command like `glpi-project/coding-standard:0.5`. Refer to the [coding-standard project changelog](#) to know more ;)

You can then for example add a line in your `.travis.yml` file to automate checking:

```
script:
  - vendor/bin/phpcs -p --ignore=vendor --standard=vendor/glpi-project/coding-standard/
  ↪ GlpiStandard/ .
```

Note: Coding standards and theirs checks are enabled per default using the [empty plugin facilities](#)



5.3 Database

Warning: A plugin should **never** change core's database! It just add its own tables to manage its own data.

Of course, plugins rely on *GLPI database model* and must therefore respect *database naming conventions*.

Creating, updating or removing tables is done by the plugin, at installation, update or uninstallation; functions added in the `hook.php` file will be used for that; and you will rely on the `Migration` class provided from GLPI core. Please refer to this documentation to know more about various *Migration* possibilities.

5.3.1 Creating and updating tables

Creating and updating tables must be done in the plugin installation process. You will add the required code to the `plugin_{myplugin}_install`. As the same function is used for both installation and update, you'll have to make tests to know what to do.

For example, we will create a basic table to store some configuration for our plugin:

```
<?php
/**
 * Install hook
 *
 * @return boolean
 */
function plugin_myexample_install() {
    global $DB;

    //instanciate migration with version
    $migration = new Migration(100);

    //Create table only if it does not exists yet!
    if (!$DB->tableExists('glpi_plugin_myexample_configs')) {
        //table creation query
        $query = "CREATE TABLE `glpi_plugin_myexample_config` (
            `id` INT(11) NOT NULL autoincrement,
            `name` VARCHAR(255) NOT NULL,
            PRIMARY KEY (`id`)
        ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci ROW_
        ↪FORMAT=DYNAMIC";
        $DB->queryOrDie($query, $DB->error());
    }

    //execute the whole migration
    $migration->executeMigration();

    return true;
}
```

The update part is quite the same. Considering our previous example, we missed to add a field in the configuration table to store the config value; and we should add an index on the `name` column. The code will become:

```
<?php
/**
 * Install hook
 *
 * @return boolean
 */
function plugin_myexample_install() {
    global $DB;

    //instanciate migration with version
    $migration = new Migration(100);
```

(continues on next page)

(continued from previous page)

```

//Create table only if it does not exists yet!
if (!$DB->tableExists('glpi_plugin_myexample_configs')) {
    //table creation query
    $query = "CREATE TABLE `glpi_plugin_myexample_configs` (
        `id` INT(11) NOT NULL autoincrement,
        `name` VARCHAR(255) NOT NULL,
        PRIMARY KEY (`id`)
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci ROW_
    ↪FORMAT=DYNAMIC";
    $DB->queryOrDie($query, $DB->error());
}

if ($DB->tableExists('glpi_plugin_myexample_configs')) {
    //missed value for configuration
    $migration->addField(
        'glpi_plugin_myexample_configs',
        'value',
        'string'
    );

    $migration->addKey(
        'glpi_plugin_myexample_configs',
        'name'
    );
}

//execute the whole migration
$migration->executeMigration();

return true;
}

```

Of course, we can also add or remove tables in our upgrade process, drop fields, keys, ... Well, do just what you need to do :-)

5.3.2 Deleting tables

You will have to drop all plugins tables when it will be uninstalled. Just put your code into the `plugin_{myplugin}_uninstall` function:

```

<?php
/**
 * Uninstall hook
 *
 * @return boolean
 */
function plugin_myexample_uninstall() {
    global $DB;

    $tables = [
        'configs'
    ];
}

```

(continues on next page)

(continued from previous page)

```

];

foreach ($tables as $table) {
    $tablename = 'glpi_plugin_myexample_' . $table;
    //Create table only if it does not exists yet!
    if ($SDB->tableExists($tablename)) {
        $SDB->queryOrDie(
            "DROP TABLE `{$tablename}`",
            $SDB->error()
        );
    }
}

return true;
}

```



5.4 Adding and managing objects

In most of the cases, your plugin will have to manage several objects

5.4.1 Define an object

Objects definitions will be stored into the `inc/` or `src/` directory of your plugin. It is recommended to place all class files in the `src` if possible. As of GLPI 10.0, namespaces should be supported in almost all cases. Therefore, it is recommended to use namespaces for your plugin classes. For example, if your plugin is `MyExamplePlugin`, you should use the `GlpPlugin\Myexampleplugin` namespace. Note that the plugin name part of the namespace must be lowercase with the exception of the first letter. Child namespaces of `GlpPlugin\Myexampleplugin` do not need to follow this rule.

Depending on where your class files are stored, the naming convention will be different: - `inc`: File name will be the name of your class, lowercase; the class name will be the concatenation of your plugin name and your class name.

For example, if you want to create the `MyObject` in `MyExamplePlugin`; you will create the `inc/myobject.class.php` file; and the class name will be `MyExamplePluginMyObject`.

- `src`: File name will match the name of your class exactly. The class name should not be prefixed by your plugin name when using namespaces. Namespaces are supported and can be reflected as subfolders. For example, if your class is `GlpPlugin\Myexampleplugin\NS\MyObject`, the file will be `src/NS/MyObject.php`.

Your object will extends one of the *common core types* (`CommonDBTM` in our example).

Extra operations are aslo described in the *tips and tricks page*, you may want to take a look at it.

5.4.2 Add a front for my object (CRUD)

The goal is to build CRUD (Create, Read, Update, Delete) and list views for your object.

You will need:

- a class for your object (src/MyObject.php),
- a front file to handle display (front/myobject.php),
- a front file to handle form display (front/myobject.form.php).

First, create the src/MyObject.php file that looks like:

```
<?php
namespace GlpiPlugin\Myexampleplugin;

class MyObject extends CommonDBTM {
    public function showForm($ID, array $options = []) {
        global $CFG_GLPI;

        $this->initForm($ID, $options);
        $this->showFormHeader($options);

        if (!isset($options['display'])) {
            //display per default
            $options['display'] = true;
        }

        $params = $options;
        //do not display called elements per default; they'll be displayed or returned here
        $params['display'] = false;

        $out = '<tr>';
        $out .= '<th>' . __('My label', 'myexampleplugin') . '</th>'

        $objectName = autoName(
            $this->fields["name"],
            "name",
            (isset($options['withtemplate']) && $options['withtemplate']==2),
            $this->getType(),
            $this->fields["entities_id"]
        );

        $out .= '<td>';
        $out .= Html::autocompleteTextField(
            $this,
            'name',
            [
                'value'      => $objectName,
                'display'    => false
            ]
        );
        $out .= '</td>';

        $out .= $this->showFormButtons($params);
    }
}
```

(continues on next page)

(continued from previous page)

```

        if ($options['display'] == true) {
            echo $out;
        } else {
            return $out;
        }
    }
}

```

The front/myobject.php file will be in charge to list objects. It should look like:

```

<?php
use GlpiPlugin\Myexampleplugin\MyObject;
include ("../../inc/includes.php");

// Check if plugin is activated...
$plugin = new Plugin();
if (!$plugin->isInstalled('myexampleplugin') || !$plugin->isActivated('myexampleplugin'
→')) {
    Html::displayNotFoundError();
}

//check for ACLs
if (MyObject::canView()) {
    //View is granted: display the list.

    //Add page header
    Html::header(
        __('My example plugin', 'myexampleplugin'),
        $_SERVER['PHP_SELF'],
        'assets',
        MyObject::class,
        'myobject'
    );

    Search::show(MyObject::class);

    Html::footer();
} else {
    //View is not granted.
    Html::displayRightError();
}

```

And finally, the front/myobject.form.php will be in charge of CRUD operations:

```

<?php
use GlpiPlugin\MyExamplePlugin\MyObject;
include ("../../inc/includes.php");

// Check if plugin is activated...
$plugin = new Plugin();
if (!$plugin->isInstalled('myexampleplugin') || !$plugin->isActivated('myexampleplugin

```

(continues on next page)

(continued from previous page)

```

→ ')) {
    Html::displayNotFoundError();
}

$object = new MyObject();

if (isset($_POST['add'])) {
    //Check CREATE ACL
    $object->check(-1, CREATE, $_POST);
    //Do object creation
    $newid = $object->add($_POST);
    //Redirect to newly created object form
    Html::redirect("{$_CFG_GLPI['root_doc']}/plugins/front/myobject.form.php?id=$newid");
} else if (isset($_POST['update'])) {
    //Check UPDATE ACL
    $object->check($_POST['id'], UPDATE);
    //Do object update
    $object->update($_POST);
    //Redirect to object form
    Html::back();
} else if (isset($_POST['delete'])) {
    //Check DELETE ACL
    $object->check($_POST['id'], DELETE);
    //Put object in dustbin
    $object->delete($_POST);
    //Redirect to objects list
    $object->redirectToList();
} else if (isset($_POST['purge'])) {
    //Check PURGE ACL
    $object->check($_POST['id'], PURGE);
    //Do object purge
    $object->delete($_POST, 1);
    //Redirect to objects list
    Html::redirect("{$_CFG_GLPI['root_doc']}/plugins/front/myobject.php");
} else {
    //per default, display object
    $withtemplate = (isset($_GET['withtemplate']) ? $_GET['withtemplate'] : 0);
    $object->display(
        [
            'id' => $_GET['id'],
            'withtemplate' => $withtemplate
        ]
    );
}

```



5.5 Hooks

GLPI provides a certain amount of “hooks”. Their goal is for plugins (mainly) to work on certain places of the framework; like when an item has been added, updated, deleted, ...

This page describes current existing hooks; but not the way they must be implemented from plugins. Please refer to the plugins development documentation.

5.5.1 Standards Hooks

Usage

Aside from their goals or when/where they’re called; you will see three types of different hooks. Some will receive an item as parameter, others an array of parameters, and some won’t receive anything. Basically, the way they’re declared into your plugin, and the way you’ll handle that will differ.

All hooks called are defined in the `setup.php` file of your plugin; into the `$PLUGIN_HOOKS` array. The first key is the hook name, the second your plugin name; values can be just text (to call a function declared in the `hook.php` file), or an array (to call a static method from an object):

```
<?php
//call a function
$PLUGIN_HOOKS['hook_name']['plugin_name'] = 'function_name';
//call a static method from an object
$PLUGIN_HOOKS['other_hook']['plugin_name'] = ['ObjectName', 'methodName'];
```

Without parameters

Those hooks are called without any parameters; you cannot attach them to any itemtype; basically they’ll permit you to display extra information. Let’s say you want to call the `display_login` hook, in you `setup.php` you’ll add something like:

```
<?php
$PLUGIN_HOOKS['display_login']['myPlugin'] = 'myplugin_display_login';
```

You will also have to declare the function you want to call in you `hook.php` file:

```
<?php
/**
 * Display information on login page
 *
 * @return void
 */
public function myplugin_display_login () {
    echo "That line will appear on the login page!";
}
```

The hooks that are called without parameters are: `display_central`, `post_init`, `init_session`, `change_entity`, `change_profile`, `display_login` and `add_plugin_where`.

With item as parameter

Those hooks will send you an item instance as parameter; you'll have to attach them to the itemtypes you want to apply on. Let's say you want to call the `pre_item_update` hook for *Computer* and *Phone* item types, in your `setup.php` you'll add something like:

```
<?php
$PLUGIN_HOOKS['pre_item_update']['myPlugin'] = [
    'Computer' => 'myplugin_updateitem_called',
    'Phone'    => 'myplugin_updateitem_called'
];
```

You will also have to declare the function you want to call in your `hook.php` file:

```
<?php
/**
 * Handle update item hook
 *
 * @param CommonDBTM $item Item instance
 *
 * @return void
 */
public function myplugin_updateitem_called (CommonDBTM $item) {
    //do everything you want!
    //remember that $item is passed by reference (it is an object)
    //so changes you will do here will be used by the core.
    if ($item::getType() === Computer::getType()) {
        //we're working with a computer
    } elseif ($item::getType() === Phone::getType()) {
        //we're working with a phone
    }
}
```

The hooks that are called with item as parameter are: `item_empty`, `pre_item_add`, `post_preparedadd`, `item_add`, `pre_item_update`, `item_update`, `pre_item_purge`, `pre_item_delete`, `item_purge`, `item_delete`, `pre_item_restore`, `item_restore`, `autoinventory_information`, `item_add_targets`, `item_get_events`, `item_action_targets`, `item_get_datas`.

With array of parameters

These hooks will work just as the *hooks with item as parameter* expect they will send you an array of parameters instead of only an item instance. The array will contain two entries: `item` and `options`, the first one is the item instance, the second options that have been passed:

```
<?php
/**
 * Function that handle a hook with array of parameters
 *
 * @param array $params Array of parameters
 *
 * @return void
 */
```

(continues on next page)

(continued from previous page)

```

public function myplugin_params_hook(array $params) {
    print_r($params);
    //Will display:
    //Array
    //(
    //    [item] => Computer Object
    //    (...)
    //
    //    [options] => Array
    //    (
    //        [_target] => /front/computer.form.php
    //        [id] => 1
    //        [withtemplate] =>
    //        [tabnum] => 1
    //        [itemtype] => Computer
    //    )
    //)
}

```

The hooks that are called with an array of parameters are: `post_item_form`, `pre_item_form`, `pre_show_item`, `post_show_item`, `pre_show_tab`, `post_show_tab`, `item_transfer`.

Some hooks will receive a specific array as parameter, they will be detailed below.

Unclassified

Hooks that cannot be classified in above categories :)

secured_fields

New in version 9.4.6.

An array of fields names (with table like `glpi_mytable.myfield`) that are stored using GLPI encrypting methods. This allows plugins to add some fields to the `glpi:security:changekey` command.

Warning: Plugins have to ensure crypt migration on their side is OK; and once using it, they **must** properly declare fields.

All fields that would use the key file without being listed would be unreadable after key has been changed (and stored data would stay potentially insecure).

secured_configs

New in version 9.4.6.

An array of configuration entries that are stored using GLPI encrypting methods. This allows plugins to add some entries to the `glpi:security:changekey` command.

Warning: Plugins have to ensure crypt migration on their side is OK; and once using it, they **must** properly declare fields.

All configuration entries that would use the key file without being listed would be unreadable after key has been changed (and stored data would stay potentially insecure).

add_javascript

Add javascript in **all** pages headers

New in version 9.2: Minified javascript files are checked automatically. You will just have to provide a minified file along with the original to get it used!

The name of the minified `plugin.js` file must be `plugin.min.js`

add_css

Add CSS stylesheet on **all** pages headers

New in version 9.2: Minified CSS files are checked automatically. You will just have to provide a minified file along with the original to get it used!

The name of the minified `plugin.css` file must be `plugin.min.css`

display_central

Displays something on central page

display_login

Displays something on the login page

status

Displays status

post_init

After the framework initialization

rule_matched

After a rule has matched.

This hook will receive a specific array that looks like:

```
<?php
$hook_params = [
    'sub_type' => 'an item type',
    'rule_id'  => 'rule id',
    'input'    => array(), //original input
    'output'   => array()  //output modified by rule
];
```

redefine_menus

Add, edit or remove items from the GLPI menus.

This hook will receive the current GLPI menus definition as an argument and must return the new definition.

init_session

At session initialization

change_entity

When entity is changed

change_profile

When profile is changed

pre_kanban_content

New in version 9.5.

Set or modify the content that shows before the main content in a Kanban card.

This hook will receive a specific array that looks like:

```
<?php
$hook_params = [
    'itemtype' => string, //item type that is showing the Kanban
    'items_id' => int, //ID of itemtype showing the Kanban
    'content' => string //current content shown before main content
];
```

post_kanban_content

New in version 9.5.

Set or modify the content that shows after the main content in a Kanban card.

This hook will receive a specific array that looks like:

```
<?php
$hook_params = [
    'itemtype' => string, //item type that is showing the Kanban
    'items_id' => int, //ID of itemtype showing the Kanban
    'content' => string //current content shown after main content
];
```

kanban_filters

Add new filter definitions for Kanban by itemtype.

This data is set directly in \$PLUGIN_HOOKS like:

```
<?php
$PLUGIN_HOOKS['kanban_filters']['tag'] = [
    'Ticket' => [
        'tag' => [
            'description' => _x('filters', 'If the item has a tag'),
            'supported_prefixes' => ['!']
        ],
        'tagged' => [
            'description' => _x('filters', 'If the item is tagged'),
            'supported_prefixes' => ['!']
        ]
    ],
    'Project' => [
        'tag' => [
            'description' => _x('filters', 'If the item has a tag'),
            'supported_prefixes' => ['!']
        ],
        'tagged' => [
            'description' => _x('filters', 'If the item is tagged'),
            'supported_prefixes' => ['!']
        ]
    ]
];
```

kanban_item_metadata

Set or modify the metadata for a Kanban card. This metadata isn't displayed directly but will be used by the filtering system.

This hook will receive a specific array that looks like:

```
<?php
$hook_params = [
    'itemtype' => string, //item type that is showing the Kanban
    'items_id' => int, //ID of itemtype showing the Kanban
    'metadata' => array //current metadata array
];
```

vcard_data

Add or modify data in vCards such as IM contact information

```
<?php
$hook_params = [
    'item' => CommonDBTM, //The item the vCard is for such as a User or Contact
    'data' => array, //The current vCard data for the item
];
```

filter_actors

Add or modify data actor fields provided in the right panel of ITIL objects

```
<?php
$hook_params = [
    'actors' => array, // actors array send to select2 field
    'params' => array, // actor field param
];
```

helpdesk_menu_entry

Add a link to the menu for users with the simplified interface

```
<?php
$PLUGIN_HOOKS['helpdesk_menu_entry']['example'] = 'MY_CUSTOM_LINK';
```

helpdesk_menu_entry_icon

Add an icon for the link specified by the *helpdesk_menu_entry* hook

```
<?php
$PLUGIN_HOOKS['helpdesk_menu_entry_icon']['example'] = 'fas fa-tools';
```

debug_tabs

Add one or more new tabs to the GLPI debug panel. Each tab must define a *title* and *display_callable* which is what will be called to print the tab contents.

```
<?php
$PLUGIN_HOOKS['debug_tabs']['example'] = [
    [
        'title' => 'ExampleTab',
        'display_callable' => 'ExampleClass::displayDebugTab'
    ]
];
```

post_plugin_install

Called after a plugin is installed

post_plugin_enable

Called after a plugin is enabled

post_plugin_disable

Called after a plugin is disabled

post_plugin_uninstall

Called after a plugin is uninstalled

post_plugin_clean

Called after a plugin is cleaned (removed from the database after the folder is deleted)

Items business related

Hooks that can do some business stuff on items.

item_empty

When a new (empty) item has been created. Allow to change / add fields.

post_prepareadd

Before an item has been added, after `prepareInputForAdd()` has been run, so after rule engine has ben run, allow to edit `input` property, setting it to false will stop the process.

pre_item_add

Before an item has been added, allow to edit `input` property, setting it to false will stop the process.

item_add

After adding an item, `fields` property can be used.

pre_item_update

Before an item is updated, allow to edit `input` property, setting it to false will stop the process.

item_update

While updating an item, `fields` and `updates` properties can be used.

pre_item_purge

Before an item is purged, allow to edit `input` property, setting it to false will stop the process.

item_purge

After an item is purged (not pushed to trash, see `item_delete`). The `fields` property still available.

pre_item_restore

Before an item is restored from trash.

item_restore

After an item is restored from trash.

pre_item_delete

Before an item is deleted (moved to trash), allow to edit `input` property, setting it to false will stop the process.

item_delete

After an item is moved to trash.

autoinventory_information

After an automated inventory has occurred

item_transfer

When an item is transferred from an entity to another

item_can

New in version 9.2.

Allow to restrict user rights (can't grant more right). If `right` property is set (called during `CommonDBTM::can`) changing it allow to deny evaluated access. Else (called from `Search::addDefaultWhere`) `add_where` property can be set to filter search results.

add_plugin_where

New in version 9.2.

Permit to filter search results.

Items display related

Hooks that permits to add display on items.

pre_item_form

New in version 9.1.2.

Before an item is displayed; just after the form header if any; or at the beginning of the form. Waits for a <tr>.

post_item_form

New in version 9.1.2.

After an item form has been displayed; just before the dates or the save buttons. Waits for a <tr>.

pre_show_item

Before an item is displayed

post_show_item

After an item has been displayed

pre_show_tab

Before a tab is displayed

post_show_tab

After a tab has been displayed

show_item_stats

New in version 9.2.1.

Add display from statistics tab of a item like ticket

timeline_actions

New in version 9.4.1.

Changed in version 10.0.0: The timeline action buttons were moved to the timeline footer. Some previous actions may no longer be compatible with the new timeline and will need to be adjusted.

Display new actions in the ITIL object's timeline

timeline_answer_actions

New in version 10.0.0.

Display new actions in the ITIL object's answer dropdown

show_in_timeline

New in version 10.0.0.

Display forms in the ITIL object's timeline

Notifications

Hooks that are called from notifications

item_add_targets

When a target has been added to an item

item_get_events

After notifications events have been retrieved

item_action_targets

After target addresses have been retrieved

item_get_datas

After data for template have been retrieved

add_recipient_to_target

New in version 9.4.0.

When a recipient is added to targets.

The object passed as hook method parameter will contain a property `recipient_data` which will be an array containing `itemtype` and `items_id` fields corresponding to the added target.

5.5.2 Functions hooks

Usage

Functions hooks declarations are the same than standards hooks one. The main difference is that the hook will wait as output what have been passed as argument.

```
<?php
/**
 * Handle hook function
 *
 * @param array $data Array of something (assuming that's what we're receiving!)
 *
 * @return array
 */
public function myplugin_updateitem_called ($data) {
    //do everything you want
    //return passed argument
    return $data;
}
```

Existing hooks

unlock_fields

After a fields has been unlocked. Will receive the `$_POST` array used for the call.

restrict_ldap_auth

Additional LDAP restrictions at connection. Must return a boolean. The `dn` string is passed as parameter.

undiscloseConfigValue

Permit plugin to hide fields that should not appear from the API (like configuration fields, etc). Will receive the requested fields list.

infocom

Additional infocom information oin an item. Will receive an item instance as parameter, is expected to return a table line (<tr>).

retrieve_more_field_from_ldap

Retrieve additional fields from LDAP for a user. Will receive the current fields lists, is expected to return a fields list.

retrieve_more_data_from_ldap

Retrieve additional data from LDAP for a user. Will receive current fields list, is expected to return a fields list.

display_locked_fields

To manage fields locks. Will receive an array with `item` and `header` entries. Is expected to output a table line (<tr>).

migratetypes

Item types to migrate, will receive an array of types to be updated; must return an array of item types to migrate.

5.5.3 Automatic hooks

Some hooks are automated; they'll be called if the relevant function exists in you plugin's `hook.php` file. Required function must be of the form `plugin_{plugin_name}_{hook_name}`.

MassiveActionsFieldsDisplay

Add massive actions. Will receive an array with `item` (the item type) and `options` (the search options) as input. These hook have to output its content, and to return true if there is some specific output, false otherwise.

dynamicReport

Add parameters for print. Will receive the `$_GET` array used for query. Is expected to return an array of parameters to add.

AssignToTicket

Declare types an ITIL object can be assigned to. Will receive an empty array adn is expected to return a list an array of type of the form:

```
<?php
return [
    'TypeClass' => 'label'
];
```

MassiveActions

If plugin provides massive actions (via `$PLUGIN_HOOKS['use_massive_actions']`), will pass the item type as parameter, and expect an array of additional massive actions of the form:

```
<?php
return [
    'Class::method' => 'label'
];
```

getDropDown

To declare extra dropdowns. Will not receive any parameter, and is expected to return an array of the form:

```
<?php
return [
    'Class::method' => 'label'
];
```

rulePrepareInputDataForProcess

Provide data to process rules. Will receive an array with `item` (data used to check criteria) and `params` (the parameters) keys. Is expected to retrain an array of rules.

executeActions

Actions to execute for rule. Will receive an array with `output`, `params` and `action` keys. Is expected to return an array of actions to execute.

preProcessRulePreviewResults

Todo: Write documentation for this hook.

use_rules

Todo: Write documentation for this hook. It looks a bit particular.

ruleCollectionPrepareInputDataForProcess

Prepare input data for rules collections. Will receive an array of the form:

```
<?php
array(
    'rule_itemtype' => 'name fo the rule itemtype',
    'values'        => array(
        'input' => 'input array',
        'params' => 'array of parameters'
    )
);
```

Is expected to return an array.

preProcessRuleCollectionPreviewResults

Todo: Write documentation for this hook.

ruleImportComputer_addGlobalCriteria

Add global criteria for computer import. Will receive an array of global criteria, is expected to return global criteria array.

ruleImportComputer_getSqlRestriction

Adds SQL restriction to links. Will receive an array of the form:

```
<?php
array(
    'where_entity' => 'where entity clause',
    'input'        => 'input array',
    'criteria'     => 'complex criteria array',
    'sql_where'    => 'sql where clause as string',
    'sql_from'     => 'sql from clause as string'
)
```

Is expected to return the input array modified.

getAddSearchOptions

Adds *search options*, using “old” method. Will receive item type as string, is expected to return an array of

search options.

getAddSearchOptionsNew

Adds *search options*, using “new” method. Will receive item type as string, is expected to return an **indexed** array of search options.



5.6 Automatic actions

5.6.1 Goals

Plugins may need to run automatic actions in background, or at regular interval. GLPI provides a task scheduler for itself and its plugins.

5.6.2 Implement an automatic action

A plugin must implement its automatic action the same way as GLPI does, except the method is located in a plugin’s itemtype. See *crontasks*.

5.6.3 Register an automatic action

A plugin must register its automatic action the same way as GLPI does in its upgrade process. See *crontasks*.

5.6.4 Unregister a task

GLPI unregisters tasks of a plugin when it cleans or uninstalls it.



5.7 Massive Actions

Plugins can use the core’s *massive actions* for its own itemtypes.

They just need to additionally define a hook in their init function (setup.php):

```
<?php

function plugin_init_example() {
    $PLUGIN_HOOKS['use_massive_action']['example'] = 1;
}
```

But they can also add specific massive actions to core’s itemtypes. First, in their hook.php file, they must declare a new definition into a `plugin_pluginname_MassiveActions` function, ex addition of new action for Computer:

```
<?php

function plugin_example_MassiveActions($type) {
    $actions = [];
```

(continues on next page)

(continued from previous page)

```

switch ($type) {
    case 'Computer' :
        $myclass      = PluginExampleExample;
        $action_key    = 'DoIt';
        $action_label  = __("plugin_example_DoIt", 'example');
        $actions[$myclass.MassiveAction::CLASS_ACTION_SEPARATOR.$action_key]
            = $action_label;

        break;
}
return $actions;
}

```

Next, in the class defined in the definition, we can use the `showMassiveActionsSubForm` and `processMassiveActionsForOneItemtype` in the same way as *core documentation for massive actions*:

```

<?php

class PluginExampleExample extends CommonDBTM {

    static function showMassiveActionsSubForm(MassiveAction $ma) {

        switch ($ma->getAction()) {
            case 'DoIt':
                echo __("fill the input");
                echo Html::input('myinput');
                echo Html::submit(__('Do it'), array('name' => 'massiveaction'))."</span>";

                return true;
            }
        return parent::showMassiveActionsSubForm($ma);
    }

    static function processMassiveActionsForOneItemtype(MassiveAction $ma, CommonDBTM
    ↪ $item,
                                                    array $ids) {

        global $DB;

        switch ($ma->getAction()) {
            case 'DoIt' :
                $input = $ma->getInput();

                foreach ($ids as $id) {

                    if ($item->getFromDB($id)
                        && $item->doIt($input)) {
                        $ma->itemDone($item->getType(), $id, MassiveAction::ACTION_OK);
                    } else {
                        $ma->itemDone($item->getType(), $id, MassiveAction::ACTION_KO);
                        $ma->addMessage(__("Something went wrong"));
                    }
                }
            }
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

        return;
    }
    parent::processMassiveActionsForOneItemtype($ma, $item, $ids);
}
}

```



5.8 Tips & tricks

5.8.1 Add a tab on a core object

In order to add a new tab on a core object, you will have to:

- register your class against core object(s) telling it you will add a tab,
- use `getTabNameForItem()` to give tab a name,
- use `displayTabContentForItem()` to display tab contents.

First, in the `plugin_init_{plugin_name}` function, add the following:

```

<?php
//[...]
Plugin::registerClass(
    GmpiPlugin\Myexample\MyClass::class, [
        'addtabon' => [
            'Computer',
            'Phone'
        ]
    ]
);
//[...]

```

Here, we request to add a tab on *Computer* and *Phone* objects.

Then, in your `src/MyClass.php` (in which `MyClass` is defined):

```

<?php
function getTabNameForItem(CommonGLPI $item, $withtemplate=0) {
    switch ($item::getType()) {
        case Computer::getType():
        case Phone::getType():
            return __('Tab from my plugin', 'myexampleplugin');
            break;
    }
    return '';
}

static function displayTabContentForItem(CommonGLPI $item, $tabnum=1, $withtemplate=0) {
    switch ($item::getType()) {
        case Computer::getType():

```

(continues on next page)

(continued from previous page)

```

        //display form for computers
        self::displayTabContentForComputer($item);
        break;
    case Phone::getType():
        self::displayTabContentForPhone($item);
        break;
    }
    if ($item->getType() == 'ObjetDuCoeur') {
        $monplugin = new self();
        $ID = $item->getField('id');
        // j'affiche le formulaire
        $monplugin->nomDeLaFonctionQuiAfficheraLeContenuDeMonOnglet();
    }
    return true;
}

private static function displayTabContentForComputer(Computer $item) {
    //...
}

private static function displayTabContentForPhone(Phone $item) {
    //...
}

```

On the above example, we have used two different methods to display tab, depending on item type. You could of course use only one if there is no (or minor) differences at display.

5.8.2 Add a tab on one of my plugin objects

In order to add a new tab on your plugin object, you will have to:

- use `defineTabs()` to register the new tab,
- use `getTabNameForItem()` to give tab a name,
- use `displayTabContentForItem()` to display tab contents.

Then, in your `src/MyClass.php`:

```

<?php
function defineTabs($options=array()) {
    $ong = array();
    //add main tab for current object
    $this->addDefaultFormTab($ong);
    //add core Document tab
    $this->addStandardTab(__('Document'), $ong, $options);
    return $ong;
}

/**
 * Définition du nom de l'onglet
 */

```

(continues on next page)

(continued from previous page)

```

function getTabNameForItem(CommonGLPI $item, $withtemplate=0) {
    switch ($item::getType()) {
        case __CLASS__:
            return __('My plugin', 'myexampleplugin');
            break;
    }
    return '';
}

/**
 * Définition du contenu de l'onglet
 */
static function displayTabContentForItem(CommonGLPI $item, $tabnum=1, $withtemplate=0) {
    switch ($item::getType()) {
        case __CLASS__:
            self::myStaticMethod();
            break;
    }
    return true;
}

```

5.8.3 Add several tabs

On the same model you create one tab, you may add several tabs.

```

<?php
function getTabNameForItem(CommonGLPI $item, $withtemplate=0) {
    $song = [
        __('My first tab', 'myexampleplugin'),
        __('My second tab', 'myexampleplugin')
    ];
    return $song;
}

static function displayTabContentForItem(CommonGLPI $item, $tabnum=0, $withtemplate=0) {
    switch ($tabnum) {
        case 0 : __("My first tab"
            //do something
            break;
        case 1 : __("My second tab"
            //do something else
            break;
    }
    return true;
}

```


5.8.4 Add an object in dropdowns

Just add the following to your object class (src/MyObject.php):

```
<?php
function plugin_myexampleplugin_getDropdown() {
    return [MyObject::class => MyObject::getTypeName(2)];
}
```



5.9 Notification modes

Core GLPI provides two notifications modes as of today:

- email (sends email),
- ajax (send browser notifications if/when user is logged)

It is possible to extend this mechanism in order to create another mode to use. Let's take a tour... We'll take example of a plugin designed to send SMS to the users.

5.9.1 Required configuration

A few steps are required to setup the mode. In the `init` method (setup.php file); register the mode:

```
<?php
public function plugin_init_sms {
    //[...]

    if ($plugin->isActivated('sms')) {
        Notification_NotificationTemplate::registerMode(
            Notification_NotificationTemplate::MODE_SMS, //mode itself
            __('SMS', 'plugin_sms'),                      //label
            'sms'                                           //plugin name
        );
    }

    //[...]
}
```

Note: GLPI will look for classes named like `Plugin{NAME}Notification{MODE}`.

In the above example; we have used one of the provided (but not yet used) modes from the core. If you need a mode that does not exist, you can of course create yours!

In order to make your new notification active, you will have to declare a `notifications_{MODE}` variable in the main configuration: You will add it at install time, and remove it on uninstall... In the `hook.php` file:

```
<?php

function plugin_sms_install() {
```

(continues on next page)

(continued from previous page)

```

    Config::setConfigurationValues('core', ['notifications_sms' => 0]);
    return true;
}

function plugin_sms_uninstall() {
    $config = new Config();
    $config->deleteConfigurationValues('core', ['notifications_sms']);
    return true;
}

```

5.9.2 Settings

You will probably need some configuration settings to get your notifications mode to work. You can register and retrieve additional configuration values using core Config object:

```

<?php
//set configuration
Config::setConfigurationValues(
    'plugin:sms', //context
    [ //values
        'server' => '',
        'port'   => ''
    ]
);

//get configuration
$conf = Config::getConfigurationValues('plugin:sms');
//$conf will be ['server' => '', 'port' => '']

```

That said, we need to create a class to handle the settings, and a front file to display them. The class must be named GlpiPlugin\Sms\NotificationSmsSetting and must be in the src/NotificationSmsSetting.php file. It have to extends the NotificationSetting core class :

```

<?php
namespace GlpiPlugin\Sms;
if (!defined('GLPI_ROOT')) {
    die("Sorry. You can't access this file directly");
}

/**
 * This class manages the sms notifications settings
 */
class NotificationSmsSetting extends NotificationSetting {

    static function getTypeName($nb=0) {
        return __('SMS followups configuration', 'sms');
    }

    public function getEnableLabel() {

```

(continues on next page)

(continued from previous page)

```

    return __('Enable followups via SMS', 'sms');
}

static public function getMode() {
    return Notification_NotificationTemplate::MODE_SMS;
}

function showFormConfig($options = []) {
    global $CFG_GLPI;

    $conf = Config::getConfigurationValues('plugin:sms');
    $params = [
        'display' => true
    ];
    $params = array_merge($params, $options);

    $out = "<form action='".Toolbox::getItemTypeFormURL(__CLASS__)." method='post'>";
    $out .= Html::hidden('config_context', ['value' => 'plugin:sms']);
    $out .= "<div>";
    $out .= "<input type='hidden' name='id' value='1'>";
    $out .= "<table class='tab_cadre_fixe'>";
    $out .= "<tr class='tab_bg_1'><th colspan='4'>".__n('SMS notification', 'SMS_
    notifications', Session::getPluralNumber(), 'sms')."</th></tr>";

    if ($CFG_GLPI['notifications_sms']) {
        //TODO
        $out .= "<tr><td colspan='4'>".__('SMS notifications are not implemented yet.
    ', 'sms'). "</td></tr>";
    } else {
        $out .= "<tr><td colspan='4'>".__('Notifications are disabled.') . " <a href=
    '{ $CFG_GLPI['root_doc'] }/front/setup.notification.php'>" . __('See configuration') . "
    </td></tr>";
    }
    $options['candel'] = false;
    if ($CFG_GLPI['notifications_sms']) {
        $options['addbuttons'] = array('test_sms_send' => __('Send a test SMS to you',
    'sms'));
    }

    //Ignore display parameter since showFormButtons is now ready :/ (from all but
    tests)
    echo $out;

    $this->showFormButtons($options);
}
}

```

The front form file, located at `front/notificationssmssetting.form.php` will be quite simple. It handles the display of the configuration form, update of the values, and test send (if any):

```
<?php
use Glpi\Plugin\Sms\NotificationSmsSetting;
include ('../../inc/includes.php');

Session::checkRight("config", UPDATE);
$notificationsms = new NotificationSmsSetting();

if (!empty($_POST["test_sms_send"])) {
    NotificationSmsSetting::testNotification();
    Html::back();
} else if (!empty($_POST["update"])) {
    $config = new Config();
    $config->update($_POST);
    Html::back();
}

Html::header(Notification::getTypeName(Session::getPluralNumber()), $_SERVER['PHP_SELF'],
    ↪ "config", "notification", "config");

$notificationsms->display(array('id' => 1));

Html::footer();
```

5.9.3 Event

Once the new mode has been enabled; it will try to raise core events. You will need to create an event class named `GlpiPlugin\Sms\NotificationEventSms` that implements `NotificationEventInterface` and extends `NotificationEventAbstract` in the `src/NotificationEventSms.php` file.

Methods to implement are:

- `getTargetFieldName`: defines the name of the target field;
- `getTargetField`: populates if needed the target field to use. For a SMS plugin, it would retrieve the phone number from users table for example;
- `canCron`: whether notification can be called from a crontask. For the SMS plugins, it would be true. It is set to false for ajax based events; because notifications are requested from user browser;
- `getAdminData`: as global admin is not a real user; you can define here the data used to send the notification;
- `getEntityAdminData`: same as the above, but for entities admins rather than global admin;
- `send`: method that will really send data.

The `raise` method declared in the interface is implemented in the abstract class; since it should be used as it for every mode. If you want to do extra process in the `raise` method, you should override the `extraRaise` method. This is done in the core to add signatures in the mailing for example.

Note: Notifications uses the `QueueNotification` to store its data. Each notification about to be sent will be stored in the relevant table. Rows are updated once the notification has really be send (set `is_deleted` to 1 and update `sent_time`).

An example class for SMS Events would look like the following:

```

<?php
namespace GlpiPlugin\Sms;
class NotificationEventSms implements NotificationEventInterface {

    static public function getTargetFieldName() {
        return 'phone';
    }

    static public function getTargetField(&$data) {
        $field = self::getTargetFieldName();

        if (!isset($data[$field])
            && isset($data['users_id'])) {
            // No phone set: get one for user
            $user = new user();
            $user->getFromDB($data['users_id']);

            $phone_fields = ['mobile', 'phone', 'phone2'];
            foreach ($phone_fields as $phone_field) {
                if (isset($user->fields[$phone_field]) && !empty($user->fields[
→$phone_
                field])) {
                    $data[$field] = $user->fields[$phone_field];
                    break;
                }
            }
        }

        if (!isset($data[$field])) {
            //Missing field; set to null
            $data[$field] = null;
        }

        return $field;
    }

    static public function canCron() {
        return true;
    }

    static public function getAdminData() {
        //no phone available for global admin right now
        return false;
    }

    static public function getEntityAdminsData($entity) {
        global $DB, $CFG_GLPI;

        $iterator = $DB->request([
            'FROM' => 'glpi_entities',

```

(continues on next page)

(continued from previous page)

```

        'WHERE' => ['id' => $entity]
    ]);

    $admins = [];

    while ($row = $iterator->next()) {
        $admins[] = [
            'language' => $CFG_GLPI['language'],
            'phone'     => $row['phone_number']
        ];
    }

    return $admins;
}

static public function send(array $data) {
    //data is an array of notifications to send. Process the array and send real SMS.
    ↪here!
    throw new \RuntimeException('Not yet implemented!');
}
}

```

5.9.4 Notification

Finally, create a GlpiPlugin\Sms\NotificationSms class that implements the NotificationInterface in the src/NotificationSms.php file.

Methods to implement are:

- **check**: to validate data (checking if a mail address is well formed, ...);
- **sendNotification**: to store raised event notification in the QueueNotification;
- **testNotification**: used from settings to send a test notification.

Again, the SMS example:

```

<?php
namespace GlpiPlugin\Sms;
class NotificationSms implements NotificationInterface {

    static function check($value, $options = []) {
        //Does nothing, but we could check if $value is actually what we expect as a phone.
        ↪number to send SMS.
        return true;
    }

    static function testNotification() {
        $instance = new self();
        //send a notification to current logged in user
        $instance->sendNotification([
            '_itemtype' => 'NotificationSms',

```

(continues on next page)

(continued from previous page)

```

        '_items_id'                => 1,
        '_notificationtemplates_id' => 0,
        '_entities_id'            => 0,
        'fromname'                 => 'TEST',
        'subject'                   => 'Test notification',
        'content_text'              => "Hello, this is a test notification.",
        'to'                        => Session::getLoginUserID()
    });
}

function sendNotification($options=array()) {

    $data = array();
    $data['_itemtype']              = $options['_itemtype'];
    $data['_items_id']              = $options['_items_id'];
    $data['_notificationtemplates_id'] = $options['_notificationtemplates_id'];
    $data['_entities_id']           = $options['_entities_id'];

    $data['sendername']             = $options['fromname'];

    $data['name']                   = $options['subject'];
    $data['body_text']               = $options['content_text'];
    $data['recipient']              = $options['to'];

    $data['mode'] = Notification_NotificationTemplate::MODE_SMS;

    $mailqueue = new QueuedMail();

    if (!$mailqueue->add(Toolbox::addslashes_deep($data))) {
        Session::addMessageAfterRedirect(__('Error inserting sms notification to queue',
    'sms'), true, ERROR);
        return false;
    } else {
        //TRANS to be written in logs %1$s is the to email / %2$s is the subject of the
    mail
        Toolbox::logInFile("notification",
            sprintf(__('%1$s: %2$s'),
                sprintf(__('An SMS notification to %s was added to
    queue', 'sms'),
                    $options['to']),
                    $options['subject']."\n"));
    }

    return true;
}

```



5.10 Unit Testing

5.10.1 Goals

As a plugin's complexity increases so does the possibility of a feature or bug fix breaking some other part of the plugin. For this, it is recommended that plugins have some unit tests in place to detect when expected functionality breaks.

5.10.2 Bootstrap

Next, you need to create a bootstrap file to prepare the testing environment. This file should be located at `tests/bootstrap.php`. In the bootstrap file, you need to import a few required files and set a few constants, as well as loading your plugin. Note that you must manually check prerequisites since this check is not called automatically. For example:

```
<?php
global $CFG_GLPI;

define('GLPI_ROOT', dirname(dirname(dirname(__DIR__))));
define("GLPI_CONFIG_DIR", GLPI_ROOT . "/tests");

include GLPI_ROOT . "/inc/includes.php";
include_once GLPI_ROOT . '/tests/GLPITestCase.php';
include_once GLPI_ROOT . '/tests/DbTestCase.php';

$plugin = new \Plugin();
$plugin->checkStates(true);
$plugin->getFromDBbyDir('myplugin');

if (!plugin_myplugin_check_prerequisites()) {
    echo "\nPrerequisites are not met!";
    die(1);
}

if (!$plugin->isInstalled('myplugin')) {
    $plugin->install($plugin->getID());
}

if (!$plugin->isActivated('myplugin')) {
    $plugin->activate($plugin->getID());
}
```

You must replace “myplugin” with the directory name of your plugin.

5.10.3 Unit test files

All unit tests must be placed inside the `tests/units` directory in your plugin. Each test file has to correspond to an existing class name. If your plugin has a file `inc/test.class.php` with the class name `PluginMypluginTest`, the test file must be named `PluginMypluginTest.php`.

5.10.4 Running your tests

To run your tests, go to the root of your GLPI installation and run:

```
vendor/bin/atoum -bf plugins/myplugin/tests/bootstrap.php -d plugins/myplugin/tests/
```

You must replace “myplugin” with the directory name of your plugin.

5.10.5 Real examples

The following plugins are a good example of how to implement Atoum tests:

- [JAMF Plugin for GLPI](#)
- [Fields Plugin for GLPI](#)

5.10.6 Further reading

The [Atoum documentation](#) is a good place to start if you are not familiar with unit testing or Atoum.



PACKAGING

Various Linux distributions provides packages (*deb*, *rpm*, ...) for GLPI (Debian, Mandriva, Fedora, Redhat/CentOS, ...) and for some plugins. You may want to take a look at [Remi's package for Fedora/RHEL](#) to rely on a concrete example.

Here is some information about using and creating package:

- for users to understand how GLPI is installed
- for support to understand how GLPI work on this installation
- for packagers

6.1 Sources

GLPI public tarball is designed for ends-user; it will not fit packaging requirements. For example, this tarball bundle a lot of third party libraries, it does not ships unit tests, etc.

A better candidate would be to retrieve directly a tarball from github as package source.

6.2 Filesystem Hierarchy Standard

Most distributions requires that packages follows the [FHS \(Filesystem Hierarchy Standard\)](#):

- `/etc/glpi` for configuration files: `config_db.php` and `config_db_slave.php`. Prior to 9.2 release, other files stay in `glpi/config`; beginning with 9.2, those files have been moved;
- `/usr/share/glpi` for the web pages (read only dir);
- `/var/lib/glpi/files` for GLPI data and state information (session, uploaded documents, cache, cron, plugins, ...);
- `/var/log/glpi` for various GLPI log files.

Please refer to GLPI installation documentation in order to [get GLPI paths configured](#).

6.3 Apache Configuration File

Here is a configuration file sample for the Apache web server:

```
#To access via http://servername/glpi/
Alias /glpi /usr/share/glpi

# some people prefer a simple URL like http://glpi.example.com
#<VirtualHost *:80>
# DocumentRoot /usr/share/glpi
# ServerName glpi.example.com
#</VirtualHost>

<Directory /usr/share/glpi>
    Options None
    AllowOverride None

    # to overwrite default configuration which could be less than recommended value
    php_value memory_limit 64M

    <IfModule mod_authz_core.c>
        # Apache 2.4
        Require all granted
    </IfModule>
    <IfModule !mod_authz_core.c>
        # Apache 2.2
        Order Deny,Allow
        Allow from All
    </IfModule>
</Directory>

<Directory /usr/share/glpi/install>
    # 15" should be enough for migration in most case
    php_value max_execution_time 900
    php_value memory_limit 128M
</Directory>

# This sections replace the .htaccess files provided in the tarball
<Directory /usr/share/glpi/config>
    <IfModule mod_authz_core.c>
        # Apache 2.4
        Require all denied
    </IfModule>
    <IfModule !mod_authz_core.c>
        # Apache 2.2
        Order Deny,Allow
        Deny from All
    </IfModule>
</Directory>

<Directory /usr/share/glpi/locales>
    <IfModule mod_authz_core.c>
        # Apache 2.4
```

(continues on next page)

(continued from previous page)

```

    Require all denied
</IfModule>
<IfModule !mod_authz_core.c>
    # Apache 2.2
    Order Deny,Allow
    Deny from All
</IfModule>
</Directory>

<Directory /usr/share/glpi/install/mysql>
    <IfModule mod_authz_core.c>
        # Apache 2.4
        Require all denied
    </IfModule>
    <IfModule !mod_authz_core.c>
        # Apache 2.2
        Order Deny,Allow
        Deny from All
    </IfModule>
</Directory>

<Directory /usr/share/glpi/scripts>
    <IfModule mod_authz_core.c>
        # Apache 2.4
        Require all denied
    </IfModule>
    <IfModule !mod_authz_core.c>
        # Apache 2.2
        Order Deny,Allow
        Deny from All
    </IfModule>
</Directory>

```

6.4 Logs files rotation

Here is a logrotate sample configuration file (/etc/logrotate.d/glpi):

```

# Rotate GLPI logs daily, only if not empty
# Save 14 days old logs under compressed mode
/var/log/glpi/*.log {
    daily
    rotate 14
    compress
    notifempty
    missingok
    create 644 apache apache
}

```

6.5 SELinux stuff

For SELinux enabled distributions, you need to declare the correct context for the folders.

As an example, on Redhat based distributions:

- /etc/glpi and /var/lib/glpi: httpd_sys_script_rw_t, the web server need to write the config file in the former and various data in the latter;
- /var/log/glpi: httpd_log_t (apache log type: write only, no delete).

6.6 Use system cron

GLPI provides an internal cron for automated tasks. Using a system cron allow a more consistent and regular execution, for example when no user connected on GLPI.

Note: cron.php should be run as the web server user (apache or www-data)

You will need a crontab file, and to configure GLPI to use system cron. Sample cron configuration file (/etc/cron.d/glpi):

```
# GLPI core
# Run cron from to execute task even when no user connected
*/4 * * * * apache /usr/bin/php /usr/share/glpi/front/cron.php
```

To tell GLPI it must use the system crontab, simply define the GLPI_SYSTEM_CRON constant to true in the config_path.php file:

```
<?php
//[...]

//Use system cron
define('GLPI_SYSTEM_CRON', true);
```

6.7 Using system libraries

Since most distributions prefers the use of system libraries (maintained separately); you can't rely on the vendor directory shipped in the public tarball; nor use composer.

The way to handle third party libraries is to provide an autoload file with paths to you system libraries. You'll find all requirements from the composer.json file provided along with GLPI:

```
<?php
$vendor = '##DATADIR##/php';
// Dependencies from composer.json
// "ircmaxell/password-compat"
// => useless for php >= 5.5
//require_once $vendor . '/password_compat/password.php';
// "jasig/phpcas"
require_once '##DATADIR##/pear/CAS/Autoload.php';
```

(continues on next page)

(continued from previous page)

```
// "iamcal/lib_autolink"
require_once $vendor . '/php-iamcal-lib-autolink/autoload.php';
// "phpmailer/phpmailer"
require_once $vendor . '/PHPMailer/PHPMailerAutoload.php';
// "sabre/vobject"
require_once $vendor . '/Sabre/VObject/autoload.php';
// "simplepie/simplepie"
require_once $vendor . '/php-simplepie/autoloader.php';
// "tecnickcom/tcpdf"
require_once $vendor . '/tcpdf/tcpdf.php';
// "zendframework/zend-cache"
// "zendframework/zend-i18n"
// "zendframework/zend-loader"
require_once $vendor . '/Zend/autoload.php';
// "zetacomponents/graph"
require_once $vendor . '/ezc/Graph/autoloader.php';
// "ramsey/array_column"
// => useless for php >= 5.5
// "michelf/php-markdown"
require_once $vendor . '/Michelf/markdown-autoload.php';
// "true/punycode"
if (file_exists($vendor . '/TrueBV/autoload.php')) {
    require_once $vendor . '/TrueBV/autoload.php';
} else {
    require_once $vendor . '/TrueBV/Punycode.php';
}
```

Note: In the above example, the `##DATADIR##` value will be replaced by the correct value (`/usr/share/php` for instance) from the specfile using macros. Adapt with your build system possibilities.

6.8 Using system fonts rather than bundled ones

Some distribution prefers the use of system fonts (maintained separately).

GLPI use the `FreeSans.ttf` font you can configure adding in the `config_path.php`:

```
<?php
//[...]

define('GLPI_FONT_FREESANS', '/path/to/FreeSans.ttf');
```



If you want to help us improve the current documentation, feel free to open pull requests! You can [see open issues](#) and [join the documentation mailing list](#).

Here is a list of things to be done:

Todo:

- datafields option

- difference between searchunit and delay_unit
 - dropdown translations
 - giveItem
 - export
 - fulltext search
-

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/glpi-developer-documentation/checkouts/master/source/devapi/search.rst`, line 27.)

Todo: Write documentation for this hook.

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/glpi-developer-documentation/checkouts/master/source/plugins/hooks.rst`, line 608.)

Todo: Write documentation for this hook. It looks a bit particular.

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/glpi-developer-documentation/checkouts/master/source/plugins/hooks.rst`, line 614.)

Todo: Write documentation for this hook.

(The [original entry](#) is located in `/home/docs/checkouts/readthedocs.org/user_builds/glpi-developer-documentation/checkouts/master/source/plugins/hooks.rst`, line 636.)

