GLPI Developer Documentation Documentation

Teclib'

Jun 04, 2026

CONTENTS

CHAPTER

ONE

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 exists, 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

Testing is a very important part of the development process. The reference for tests is the GitHub CI. The easier way to proceed with testing is to use the provided test script run_tests.sh which uses Docker and bootstrap everything needed.

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:

- *.devcontainer*
- *.docker*
- *.tx*: Transifex configuration
- *.github*: Github related setup (CI, builds, templates and so on)
- *ajax*
  - **.php*: Ajax components
- *bin*
  - *console*: GLPI console executable
- *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`)
  - *glpicrypt.key*: Crypt key used to encrypt/decrypt database stored passwords
- *css*
  - *...*: SCSS stylesheets
  - **.css*: SCSS stylesheets
- *dependency_injection*: dependency injection setup
- *files* Files written by GLPI or plugins (documents, session files, log files, ...)
- *front*
  - **.php*: Front components (all displayed pages)
- *inc*

  - **.php*: Functions and definitions
- *install*
  - *migrations*: Update files
  - *mysql*: MariaDB/MySQL schemas
  - **.php*: upgrades scripts, installer, data to inject after installation
- *js*
  - **.js*: Javascript source files
- *lib*
  - ...: external Javascript libraries
- *locales*
  - *glpi.pot*: Gettext's POT file
  - **.po*: Gettext's translations
  - **.mo*: Gettext's compiled translations
- *marketplace*:
  - ...: where all plugins from marketplace land
- *plugins*:
  - ...: where all plugins land
- *public*:
  - ...: images, compiled stylesheets and javascripts
  - *index.php*: application main entry point
- *resources*: Various resources
- *routes*: Routing setup
- *src*
  - ...: Classes
  - **.php*: Classes
- *stubs*
- *templates*: Twig templates files
  - *twig_components*: Symfony Twig Components (see documentation)
- *tests*: unit and integration tests
- *tools*: a bunch of tools
- *version*: Current version for internal use
- *vendor*: third party libs installed from composer (see composer.json below)

- *.gitignore*: Git ignore list
- *apirest.md*: REST API documentation
- *CHANGELOG.md*: Changes
- *LICENSE*: License file
- *composer.json*: Definition of PHP third party libraries (see composer)
- *package.json*: Definition of javascript third party libraries (see NPM)
- *phpunit.xml.dist*: unit testing configuration file
- ...: various files to setup builds, lint and so on

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 bre 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, *{you 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 you 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-sthing* 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, mays sure it has been discussed by the community. Open - if it does not exists yet - a ticket with your detailed proposition. Fo 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 no 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 PHPUnit for PHP tests.

For JavaScript tests, GLPI uses the Jest testing framework. Its documentation can be found at: https://devdocs.io/jest/.

1.7.1 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 lend to unpredictable further tests results.

1.7.2 Launch tests

All required *third party libraries are installed at once*, including testing framework.

There are several directories for tests:

- `phpunit/functionnal` for main core unit/functional tests;
- `phpunit/LDAP` - requires a LDAP server;
- `phpunit/web` - requires a web server;
- `phpunit/imap` - requires mail server;

You can choose to run tests on a whole directory, or on any file (+ on a specific method). Refer to PHPUnit documentation for more information.

```
$ ./vendor/bin/phpunit phpunit/functional/
[...]
$ ./vendor/bin/phpunit phpunit/functional/ComputerTest.php
[...]
$ $ ./vendor/bin/phpunit phpunit/functional/ComputerTest.php --filter testSomething
```

If you want to run the web tests suite, you need to run a web server, and give tests its URL when running. Here is an example using PHP native webserver:

```
$ php -S localhost:8088 tests/router.php &>/dev/null &
$ GLPI_URI=http://localhost:8088 ./vendor/bin/phpunit phpunit/web/
```

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.

CHAPTER

TWO

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	`self::theMethod()`
parent class	`parent::theMethod()`
another class	`ClassName::theMethod()`

2.2 Static or Non static?

Some methods in the source code as 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
<?php

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

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

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
<?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 `mysql_connect` 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
<?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

Linting (checking and fixing coding standards) is a good way to ensure code quality and consistency of the code base. This is done using [PHP CodeSniffer](https://github.com/squizlabs/PHP_CodeSniffer), [PHPStan](https://phpstan.org/), [ESLint](https://eslint.org/), [Stylelint](https://stylelint.io/) and [TwigCS](https://github.com/friendsoftwig/twigcs).

This can run the tasks using Docker, on your local host use *./tests/run_tests.sh lint* to proceed to all linting tasks, or use a scoped task, *./tests/run_tests.sh lint_php* to proceed to PHP linting only for example. All possible lintings are listed in the *./tests/run_tests.sh* script. For faster action you can run the scripts on your local machine or Docker development

container using *node_modules/.bin/eslint . && echo "ESLint found no errors"* for example. You can find all the commands in [*.github/actions* directory](https://github.com/glpi-project/glpi/tree/main/.github/actions).

CHAPTER

THREE

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 apsect 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 thant 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:

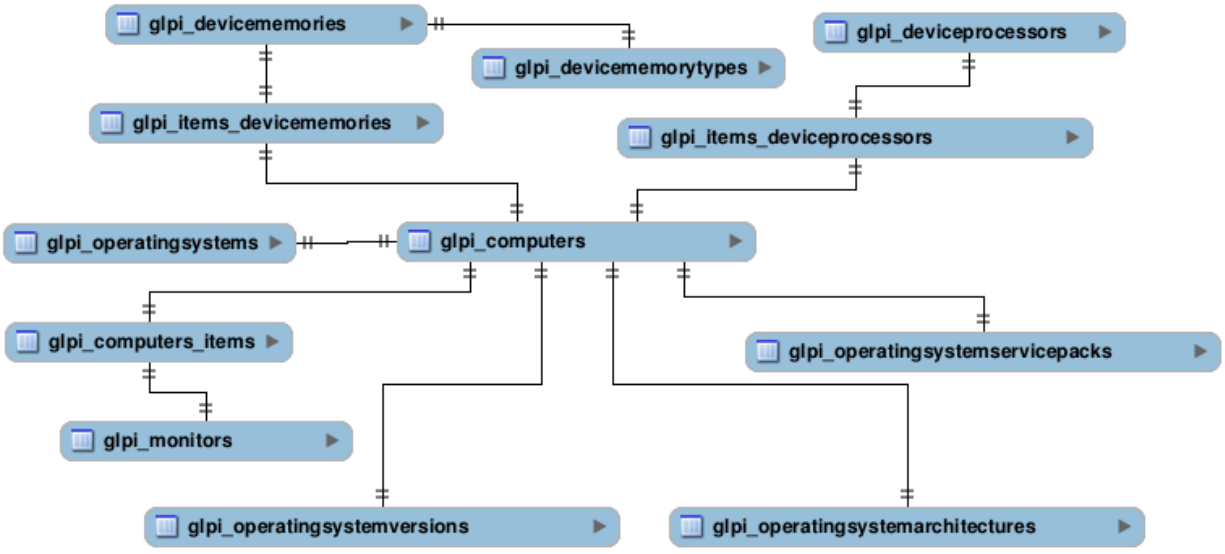
- mange ACLs,
- get and set status,
- get counts,
- do some display,
- ...

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`	`glpi_computers`
`Ticket`	`glpi_tickets`
`ITILCategory`	`glpi_itilcategories`
`PluginExampleProfile`	`glpi_plugin_example_profiles`

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
glpi_computers	computers_id
glpi_tickets	tickets_id
glpi_itilcategories	itilcategories_id
glpi_plugin_example_profiles	plugin_example_profiles_id

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 escape values to prevent SQL injections
- without having to take care of freeing resources
- iterable
- countable

Basic usage

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

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

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

Arguments

The `request` method takes as argument an array of criteria with explicit SQL clauses (`FROM`, `WHERE` and so on)

FROM clause

The SQL `FROM` clause can be a string or an array of strings:

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

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

Fields selection

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

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

(continues on next page)

(continued from previous page)

```
$DB->request(['SELECT' => '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(['SELECT' => ['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 `ON` (or the `FKEY` equivalent), to describe how to join the tables.

Left join

Using the `LEFT JOIN` option, with some criteria:

```
<?php
$DB->request([
    'FROM' => 'glpi_computers',
    'LEFT JOIN' => [
        'glpi_computerdisks' => [
            'ON' => [
                '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:

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

(continues on next page)

(continued from previous page)

```
//       INNER JOIN `glpi_computerdisks`
//         ON (`glpi_computers`.`id` = `glpi_computerdisks`.`computer_id`)
```

Right join

Using the `RIGHT JOIN` option, with some criteria:

```
<?php
$DB->request([
    'FROM' => 'glpi_computers',
    'RIGHT JOIN' => [
        'glpi_computerdisks' => [
            'ON' => [
                '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

Added 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' => [
         'ON' => [
            '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

Added 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`
```

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`
```

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
```

(continues on next page)

(continued from previous page)

```
↪']]);
// => 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']],
        ['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'))
```

Criteria unicity

Indexed array entries must be unique; otherwise PHP will only take the last one. The following example is incorrect:

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

The right way would be to enclose each condition in another array, like:

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

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

Added 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

Added 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'
   ]
```

(continues on next page)

(continued from previous page)

```
]);
$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`
```

Added 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'
```

Added 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' => [
         'ON' => [
            '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
```

Warning

If you really cannot use any of the above, you still can make raw SQL queries:

```
<?php
$DB->doQuery('SHOW COLUMNS FROM ' . $DB::quoteName('glpi_computers'));
```

You have to ensure the query is proprely escaped!

3.2.3 Updating

Added 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'
   ]
);
// => 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.

Added 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);
// [...]
```


(cc) BY-NC-ND

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.
- `getDatas()`: 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);
```

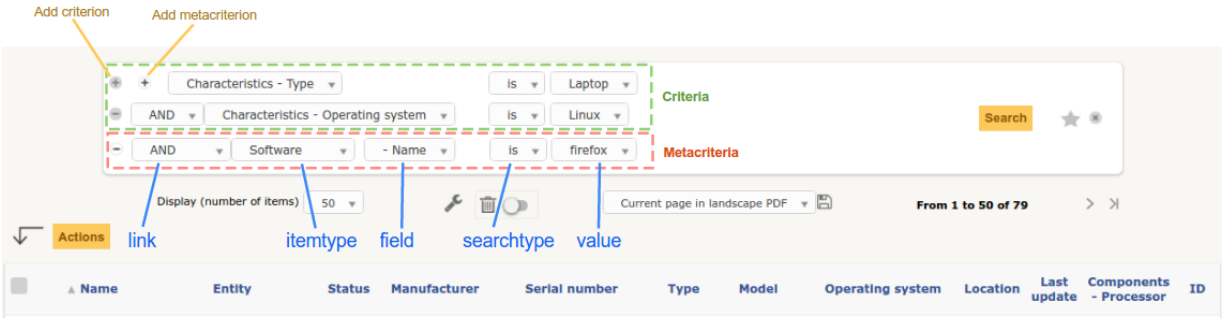
If you want to display only the multi-criteria form (with some additional options):

```
<?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 optionals.

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

   $tab[] = [
      'id'                 => 'common',
      'name'               => __('Characteristics')
   ];

   $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
<?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. Used as field name for massive action update. `field` is still used to list itemtypes property.

`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.

condition
Defines a restriction for items to choose from in filter. Do not confuse with the `condition` key in *joinparams*.

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
<?php

function getSearchOptionsMain() {
   $tab = [];

   ...

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

   ...

   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);
```

(continues on next page)

(continued from previous page)

```
}

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) {
```

(continues on next page)

(continued from previous page)

```
      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 Controllers

You need a *Controller* any time you want an URL access.

> **Note**
>
> *Controllers* are the modern way that replace all files previousely present in `front/` and `ajax/` directories.

> **Warning**
>
> Currently, not all existing `front/` or `ajax/` files have been migrated to *Controllers*, mainly because of specific behaviors or lack of time to work on migrating them.
>
> Any new feature added to GLPI >=11 **must** use *Controllers*.
>
> For plugin development, please read the *plugin-specific implementation*.

3.4.1 Creating a controller

Minimal requirements to have a working controller:

- The controller file must be placed in the `src/Glpi/Controller/` folder.
- The name of the controller must end with `Controller`.
- The controller must extends the `Glpi\Controller\AbstractController` class.
- The controller must define a route using the Route attribute.
- The controller must return some kind of response.

Example:

```
# src/Controller/Form/TagsListController.php
<?php

namespace Glpi\Controller\Form;

use Glpi\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpKernel\Exception\AccessDeniedHttpException;
use Symfony\Component\Routing\Attribute\Route;

final class TagsListController extends AbstractController
{
    #[Route(
        "/Form/TagsList",
        name: "glpi_form_tags_list",
        methods: "GET"
    )]
    public function __invoke(Request $request): Response
    {
        if (!Form::canUpdate()) {
            throw new AccessDeniedHttpException();
        }

        $tag_manager = new FormTagsManager();
        $filter = $request->query->getString('filter');

        return new JsonResponse($tag_manager->getTags($filter));
    }
}
```

3.4.2 Routing

Routing is done with the `Symfony\Component\Routing\Attribute\Route` attribute. Read more from Symfony Routing documentation.

Basic route

```
#[Symfony\Component\Routing\Attribute\Route("/my/route/url", name: "glpi_my_route_name")]
```

Dynamic route parameter

```
#[Symfony\Component\Routing\Attribute\Route("/Ticket/{id}", name: "glpi_ticket")]
```

Restricting a route to a specific HTTP method

```
#[Symfony\Component\Routing\Attribute\Route("/Tickets", name: "glpi_tickets", methods:
↪"GET")]
```

Known limitation for ajax routes

Prior to GLPI 12, if an ajax route will be accessed by multiple POST requests without a page reload then you will run into CRSF issues.

This is because GLPI's solution for this is to check a special CRSF token that is valid for multiples requests, but this special token is only checked if your url start with `/ajax`.

You will thus need to prefix your route by `/ajax` until we find a better way to handle this.

3.4.3 Reading query parameters

These parameters are found in the `$request` object:

- `$request->query` for `$_GET`
- `$request->request` for `$_POST`
- `$request->files` for `$_FILES`

Read more from Symfony Request documentation

Reading a string parameter from $_GET

```php
<?php
public function __invoke(Symfony\Component\HttpFoundation\Request $request): Response
{
    $filter = $request->query->getString('filter');
}
```

Reading an integer parameter from $_POST

```php
<?php
public function __invoke(Symfony\Component\HttpFoundation\Request $request): Response
{
    $my_int = $request->request->getInt('my_int');
}
```

Reading an array of values from $_POST

```php
<?php
public function __invoke(Symfony\Component\HttpFoundation\Request $request): Response
{
    $ids = $request->request->get("ids", []);
}
```

Reading a file

```php
<?php
public function __invoke(Symfony\Component\HttpFoundation\Request $request): Response
{
    // @var \Symfony\Component\HttpFoundation\File\UploadedFile $file
    $file = $request->files->get('my_file_input_name');
    $content = $file->getContent();
}
```

Single vs multi action controllers

The examples in this documentation use the magic `__invoke` method to force the controller to have only one action (see https://symfony.com/doc/current/controller/service.html#invokable-controllers).

In general, this is a recommended way to proceed but we do not force it and you are allowed to use multi actions controllers if you need them, by adding another public method and configuring it with the `#[Route(...)]` attribute.

Handling errors (missing rights, bad request, ...)

A controller may throw some exceptions if it receive an invalid request. Exceptions will automatically converted to error pages.

If you need exceptions with specific HTTP codes (like 4xx or 5xx codes), you can use any exception that extends `Symfony\Component\HttpKernel\Exception\HttpException`.

GLPI also provide some custom Http exceptions in the `Glpi\Exception\Http\` namespace.

Missing rights

```php
<?php
public function __invoke(Symfony\Component\HttpFoundation\Request $request): Response
{
    if (!Form::canUpdate()) {
        throw new \Glpi\Exception\Http\AccessDeniedHttpException();
    }
}
```

Invalid header

```php
<?php
public function __invoke(Symfony\Component\HttpFoundation\Request $request): Response
{
    if ($request->headers->get('Content-Type') !== 'application/json') {
        throw new \Symfony\Component\HttpKernel\Exception\
↪UnsupportedMediaTypeHttpException();
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Invalid input

```
<?php
public function __invoke(Symfony\Component\HttpFoundation\Request $request): Response
{
    $id = $request->request->getInt('id');
    if ($id == 0) {
        throw new \Glpi\Exception\Http\BadRequestHttpException();
    }
}
```

CSRF protection

Prior to GLPI 12, a form input is required in the form of

```
<input type="hidden" name="_glpi_csrf_token" value="{{ csrf_token() }}">``.
```

Starting with GLPI 12, CSRF protection is handled using Fetch metadata headers sent by client's browser. No more token form inputs are needed, you just don't need to worry about it anymore. For further information about CSRF, read MDN documentation.

In GLPI 11 or 12, csrf/tokens are checked in the CheckCsrfListener.

3.4.4 Firewall

By default, the GLPI firewall will not allow unauthenticated user to access your routes. You can change the firewall strategy with the `Glpi\Security\Attribute\SecurityStrategy` attribute.

```
<?php
#[Glpi\Security\Attribute\SecurityStrategy(Glpi\Http\Firewall::STRATEGY_NO_CHECK)]
public function __invoke(Symfony\Component\HttpFoundation\Request $request): Response
```

3.4.5 Possible responses

You may use different responses classes depending on what your controller is doing (sending json content, outputting a file, ...).

There is also a render helper method that helps you return a rendered Twig template as a Response object.

Sending JSON

```
<?php
return new Symfony\Component\HttpFoundation\JsonResponse(['name' => 'John', 'age' =>␣
↪67]);
```

Sending a file from memory

```php
<?php
$filename = "my_file.txt";
$file_content = "my file content";

$disposition = Symfony\Component\HttpFoundation\HeaderUtils::makeDisposition(
    HeaderUtils::DISPOSITION_ATTACHMENT,
    $filename,
);

$response = new Symfony\Component\HttpFoundation\Response($file_content);
$response->headers->set('Content-Disposition', $disposition);
$response->headers->set('Content-Type', 'text/plain');
return $response
```

Sending a file from disk

```php
<?php
$file_path = 'path/to/file.txt';
return new Symfony\Component\HttpFoundation\BinaryFileResponse($file_path);
```

Displaying a twig template

```php
<?php
return $this->render('path/to/my/template.html.twig', [
    'parameter_1' => 'value_1',
    'parameter_2' => 'value_2',
]);
```

Redirection

```php
<?php
return new Symfony\Component\HttpFoundation\RedirectResponse($url);
```

3.4.6 General best practices

Use thin controllers

Controller should be *thin*, which mean they should contain the minimal code needed to *glue* together the pieces of GLPI needed to handle the request.

A good controller does only the following actions:

- Check the rights
- Validate the request
- Extract what it needs from the request
- Call some methods from a dedicated service class that can process the data (using DI in the future, not possible at this time)
- Return a `Response` object

Most of the time, this will take between 5 and 15 instructions, resulting in a small method.

Make your controller final

Unless you are making a generic controller that is explicitly made to be extended, set your controller as `final`.

```
<?php
public class ApiController
final public class ApiController
```

Always restrict the HTTP method

If your controller is only meant to be used with a specific HTTP method (e.g. *POST*), it is best to define it in the `Route` attribute.

It helps others developers understand how this route must be used and help debugging when misusing the route.

```
<?php
#[Route("/my_route", name: "glpi_my_route")]
#[Route("/my_route", name: "glpi_my_route", methods: "GET")]
```

Use uppercase first route names

Since our routes will refer to GLPI itemtypes which contains upper cases letters, it is probably clearer to use *uppercase first* names for all our routes.

```
<?php
/ticket/timeline
/Ticket/Timeline
```

URL generation

Ideally, URLs should not be hard-coded but should instead be generated using their route names.

In your Controllers, you can inject the Symfony router in the constructor in order to generate URLs based on route names:

```
<?php
namespace Glpi\Controller\Custom;

use Glpi\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Attribute\Route;
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

class MyController extends AbstractController
{
    public function __construct(
        private readonly UrlGeneratorInterface $router
    ) {
    }

    public function __invoke(Request $request): Response
    {
        $route_name = $this->router->generate('my_route');
```

(continues on next page)

(continued from previous page)

```
        // ...
    }
}
```

You can also do it in Twig templates, using the `url()` or `path()` functions:

```
{{ path('my_route') }} {# Shows the url like "/my_route" #}
{{ url('my_route') }} {# Shows the url like "http://localhost/my_route" #}
```

Check out the Symfony documentation for more details about these functions:

- `url()` https://symfony.com/doc/current/reference/twig_reference.html#url
- `path()` https://symfony.com/doc/current/reference/twig_reference.html#path

3.5 Symfony Twig Components

Added in version 12.0.

Twig Components is a Symfony UX bundle allowing to use components in Twig template, inspired by HTML components. It aims to replace Twig macros.

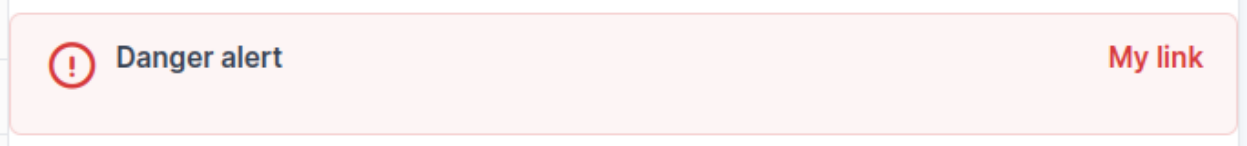
It also enables a cleaner integration with a `Vue.js`-like syntax, making components easier to maintain and review compared to the legacy macro-based integration.

The following components are available:

3.5.1 Alert

Added in version 12.0.0.

Renders an alert box (also known as callout) in the HTML.



Props

All props are optional.

- `type` **string**.
  - Possible values: `info` (default), `success`, `warning`, `danger`.
- `title` **string**.
- `message` **string**. The alert message.
- `icon` **string**. A CSS icon class, for example `ti ti-info-circle`.
  - If not set, the icon is automatically determined from the alert type.
- `important` **bool**. When `true`, the alert is visually highlighted.
  - Default: `false`.

- `link_text` **string**. Alert link, either internal or external.
- `link_url` **string**. Text for the link. If not defined will display the `link_text`
- `link_blank` **bool**. If true link target will be `_blank`, `_self` otherwise
  - Default: `true`.

Blocks

title

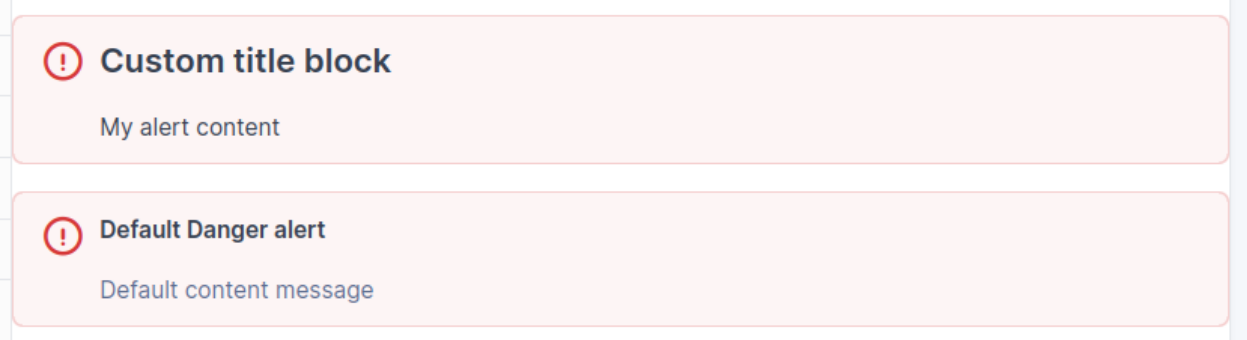
Completely overrides the title, including the wrapping <h4> element.

content

Completely overrides the message area.

```
<twig:Alert:Danger>
    <twig:block name="title">
        <h2 class="alert-title">
            Custom title block
        </h2>
    </twig:block>

    <div>
        My alert content
    </div>
</twig:Alert:Danger>
```

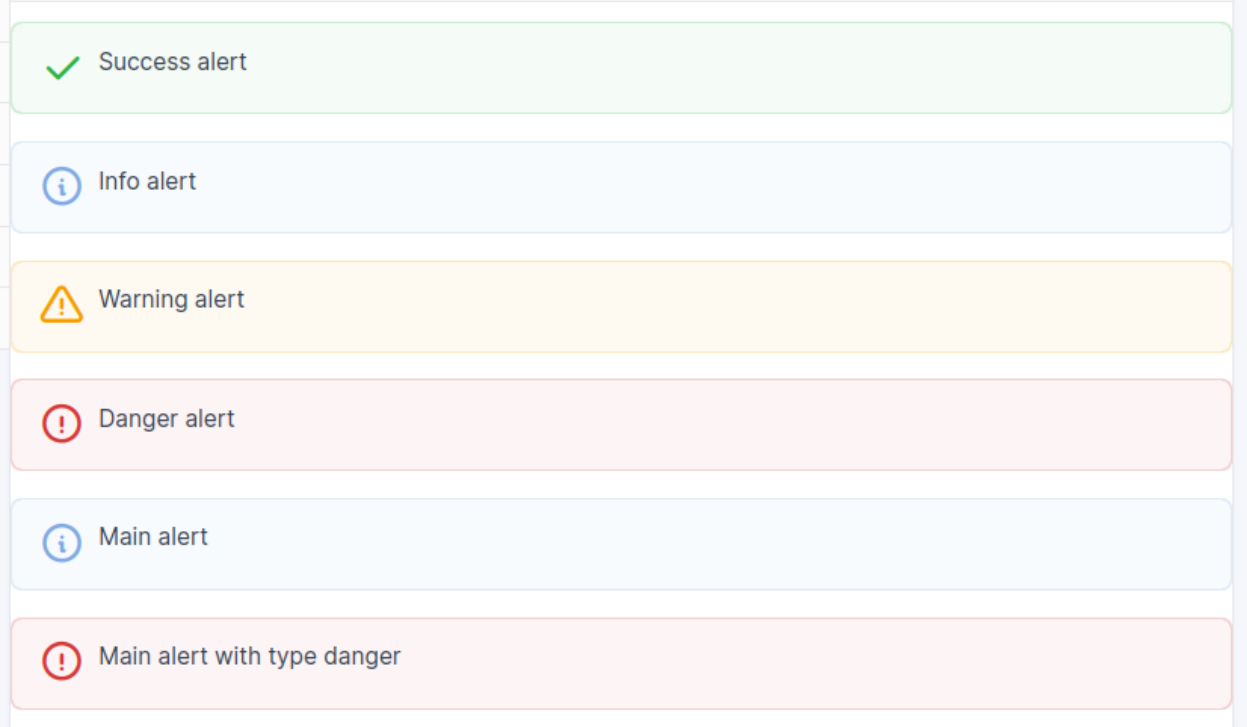


Variants

Pre-typed variant components are available as shortcuts:

```
<twig:Alert:Success>Success alert</twig:Alert:Success>
<twig:Alert:Info>Info alert</twig:Alert:Info>
<twig:Alert:Warning>Warning alert</twig:Alert:Warning>
<twig:Alert:Danger>Danger alert</twig:Alert:Danger>

<twig:Alert>Main alert</twig:Alert>
<twig:Alert type="danger">Main alert with type danger</twig:Alert>
```



3.5.2 Usage

Twig components support various integration modes. We recommend using the **Component HTML Syntax**.

Component HTML Syntax

Symfony Documentation

```
<twig:Alert title="My alert title" messages="My message" />
```

This syntax resembles modern frontend frameworks.

To pass dynamic values such as variables, booleans, or arrays, prefix the prop name with `:` and use a Twig expression:

```
<twig:Alert title="Overridden title" :messages="['Message 1', 'Message 2']" type="danger
↪" :important="true">
    <twig:block name="title">
        <h4 class="alert-title">
            Custom title block
        </h4>
        {{ parent() }} {# Renders the parent content - here: "Overridden title" #}
    </twig:block>
</twig:Alert>
```

Most components also support a default `content` block. To inject content into it, place your markup directly inside the `<twig:xx>` tag:

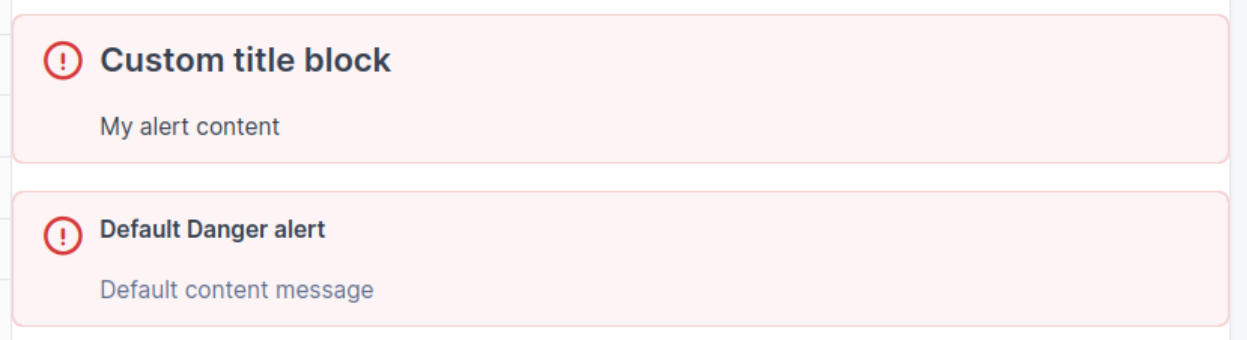
```
<twig:Alert:Danger>
    <twig:block name="title">
```

(continues on next page)

(continued from previous page)

```
        <h2 class="alert-title">
            Custom title block
        </h2>
    </twig:block>

    <div>
        My alert content
    </div>
</twig:Alert:Danger>
```



3.5.3 Creating a Component

Symfony Documentation

A Twig component consists of two parts: a **PHP class** that declares the props and logic, and a **Twig template** that defines the markup.

The PHP Class

Create a class under `src/Twig/Components/` and annotate it with `#[AsTwigComponent]`.

By default, **Public properties** become the component's props and are automatically available as variables in the template.

Public methods are also accessible from the template via the special `this` variable.

```php
<?php
namespace Twig\Components;

use Symfony\UX\TwigComponent\Attribute\AsTwigComponent;

#[AsTwigComponent(name: 'MyComponent', template: 'twig_components/MyComponent.html.twig
↪')]
class MyComponent
{
    public string $title = '';
    public bool $important = false;

    public function getComputedClass(): string
    {
        return $this->important ? 'text-bold' : '';
    }
}
```

The `name` parameter sets the tag name used in templates (`<twig:MyComponent />`). If omitted, it is derived from the class namespace relative to `Twig\Components`.

The `template` parameter is also optional. If omitted, Symfony derives the template path from the class namespace, resolved under `templates/twig_components/`.

> **Note**
>
> For components with multiple variants (e.g., `Alert:Success`, `Alert:Danger`), the recommended pattern is to extract shared props and logic into an abstract base class, then create lightweight variant classes that extend it and override the relevant defaults. See `src/Twig/Components/Alert/` (Github) for a real-world example.

The Twig Template

Place templates under `templates/twig_components/`. Props are available directly as template variables. The component object itself is accessible via `this`, which is useful for calling methods:

```
<div class="{{ this.computedClass }}">
    {% block title %}
        {% if title|length %}
            <h4>{{ title }}</h4>
        {% endif %}
    {% endblock %}

    {% block content %}{% endblock %}
</div>
```

Define `{% block %}` sections for any part of the markup that consumers may need to override.

The `{% block content %}` block is special: any markup placed directly inside the component tag (without an explicit `<twig:block>`) is injected into it automatically:

```
<twig:Alert>This text is injected into the content block.</twig:Alert>
```

Variants

Variant components share a base class and, typically, the same template. The class name determines the component tag name: a class `Twig\Components\Alert\Danger` automatically resolves to the tag `<twig:Alert:Danger>`.

```
<?php
namespace Twig\Components\Alert;

use Symfony\UX\TwigComponent\Attribute\AsTwigComponent;

#[AsTwigComponent(template: 'twig_components/Alert/Info.html.twig')]
final class Danger extends AbstractAlert
{
    public string $type = 'danger';
}
```

The `template` parameter is specified explicitly here because all Alert variants share a single template file (`twig_components/Alert/Info.html.twig`).

Testing

Two levels of tests are recommended:

- **Unit tests**: instantiate the PHP class directly and assert prop defaults and method return values. No GLPI environment needed.
- **Rendering tests**: render a Twig string using `TemplateRenderer::getInstance()->renderFromStringTemplate()` and assert the resulting HTML. These extend `GLPITestCase`.

Tests live in `tests/functional/Twig/Components/` (GitHub). See `AlertTest.php` and `AlertRenderingTest.php` for examples.

Debugging

To list all registered components and their resolved template paths, run:

```
bin/console symfony:debug:twig-component

# Using Makefile
make console c='symfony:debug:twig-component'
```


3.6 High-Level API

The High-Level API (HL API) is a new API system provided in GLPI starting with version 11.0. While the user experience is more simplified than the legacy API (the REST API available in previous versions), the implementation is quite a bit more complex. The following sections explain the various components of the new API. These sections are sorted by the recommended reading order. It is recommended that you read the High-Level API user documentation first if you have no experience with the API at all.

3.6.1 Schemas

Schemas are the definitions of the various item types in GLPI, or facades, for how they are exposed to the API. In the legacy API, all classes that extend `CommonDBTM` were exposed along with all of their search options. This is not the case with the High-Level API.

Schema Format

The schemas loosely follow the OpenAPI 3 specification to make it easier to implement the Swagger UI documentation tool. GLPI utilizes multiple custom extension fields (fields starting with 'x-') in schemas to enable advanced behavior. Schemas are defined in an array with their name as the key and definition as the value.

There exists the `\Glpi\API\HL\Doc\Schema` class which is used to represent a schema definition in some cases, but also provides constants and static methods for working with schema arrays. This includes constants for the supported property types and formats.

Let's look at a partial version of the schema definition for a User since it shows most of the possibilities:

```
'User' => [
    'x-version-introduced' => '2.0.0',
    'x-itemtype' => User::class,
    'type' => Doc\Schema::TYPE_OBJECT,
    'x-rights-conditions' => [ // Object-level extra permissions
        'read' => static function () {
            if (!\Session::canViewAllEntities()) {
```

(continues on next page)

(continued from previous page)

```
                return [
                    'LEFT JOIN' => [
                        'glpi_profiles_users' => [
                            'ON' => [
                                'glpi_profiles_users' => 'users_id',
                                'glpi_users' => 'id'
                            ]
                        ]
                    ],
                    'WHERE' => [
                        'glpi_profiles_users.entities_id' => $_SESSION[
↪'glpiactiveentities']
                    ]
                ];
            }
            return true;
        }
    ],
    'properties' => [
        'id' => [
            'type' => Doc\Schema::TYPE_INTEGER,
            'format' => Doc\Schema::FORMAT_INTEGER_INT64,
            'description' => 'ID',
            'readOnly' => true,
        ],
        'username' => [
            'x-field' => 'name',
            'type' => Doc\Schema::TYPE_STRING,
            'description' => 'Username',
        ],
        'realname' => [
            'type' => Doc\Schema::TYPE_STRING,
            'description' => 'Real name',
        ],
        'emails' => [
            'type' => Doc\Schema::TYPE_ARRAY,
            'description' => 'Email addresses',
            'items' => [
                'type' => Doc\Schema::TYPE_OBJECT,
                'x-full-schema' => 'EmailAddress',
                'x-join' => [
                    'table' => 'glpi_useremails',
                    'fkey' => 'id',
                    'field' => 'users_id',
                    'x-primary-property' => 'id' // Help the search engine understand␣
↪the 'id' property is this object's primary key since the fkey and field params are␣
↪reversed for this join.
                ],
                'properties' => [
                    'id' => [
                        'type' => Doc\Schema::TYPE_INTEGER,
                        'format' => Doc\Schema::FORMAT_INTEGER_INT64,
```

(continues on next page)

(continued from previous page)

```
                        'description' => 'ID',
                    ],
                    'email' => [
                        'type' => Doc\Schema::TYPE_STRING,
                        'description' => 'Email address',
                    ],
                    'is_default' => [
                        'type' => Doc\Schema::TYPE_BOOLEAN,
                        'description' => 'Is default',
                    ],
                    'is_dynamic' => [
                        'type' => Doc\Schema::TYPE_BOOLEAN,
                        'description' => 'Is dynamic',
                    ],
                ]
            ]
        ],
        'password' => [
            'type' => Doc\Schema::TYPE_STRING,
            'format' => Doc\Schema::FORMAT_STRING_PASSWORD,
            'description' => 'Password',
            'writeOnly' => true,
        ],
        'password2' => [
            'type' => Doc\Schema::TYPE_STRING,
            'format' => Doc\Schema::FORMAT_STRING_PASSWORD,
            'description' => 'Password confirmation',
            'writeOnly' => true,
        ],
        'picture' => [
            'type' => Doc\Schema::TYPE_STRING,
            'x-mapped-from' => 'picture',
            'x-mapper' => static function ($v) {
                global $CFG_GLPI;
                $path = \Toolbox::getPictureUrl($v, false);
                if (!empty($path)) {
                    return $path;
                }
                return $CFG_GLPI["root_doc"] . '/pics/picture.png';
            }
        ]
    ]
]
```

The first property in the definition, 'x-itemtype' is used to link the schema with an actual GLPI class. This is used to determine which table to use to access direct properties and access more data like entity restrictions and extra visiblity restrictions (when implementing the `ExtraVisibilityCriteria` class). This property is required.

Next, is a 'type' property which is part of the standard OpenAPI specification. In this case, it defines a User as an object. In general, all schemas would be objects.

Third, is an 'x-rights-conditions' property which defines special visiblity restrictions. This property may be excluded if there are no special restrictions. Currently, only 'read' restrictions can be defined here. Each type of restriction must be a callable that returns an array of criteria, or just an array of criteria, in the format used by `DBmysqlIterator`. If

the criteria is reliant on data from a session or is expensive, it should use a callable so that the criteria is resolved only at the time it is needed.

Finally, the 'properties' are defined. Each property has its unique name as the key and the definition as the value in the array. Property names do not have to match the name of the column in the database. You can specify a different column name using an 'x-field' field; Each property must have an OpenAPI 'type' defined. They may optionally define a specific 'format'. If no 'format' is specified, the generic format for that type will be used. For example, a type of `Doc\Schema::TYPE_STRING` will default to the `Doc\Schema::FORMAT_STRING_STRING` format. Properties may also optionally define a description for that property.

In this example, the 'emails' property actually refers to multiple email addresses associated with the user. The 'type' in this case is `Doc\Schema::TYPE_ARRAY`. The schema for the individual items in defined inside the 'items' property. Of course, email addresses are not stored in the same database table as users and are their own item type `EmailAddress`. Therefore, 'emails' is considered a joined object property. In joined objects, we specify which properties will be included in the data but that can be a subset of the properties of the full schema (see *Partial vs Full Schema*). The full schema can be specified using the 'x-full-schema' field. The criteria for the join is specified in the 'x-join' field (more on that in the *Joins section*).

Users have two password fields which we would never want to show via the API, but we do want them to exist in the schema to allow setting/resetting a password. In this case, both 'password' and 'password2' have a 'writeOnly' field present and set to true.

The last property shown, 'picture', is an example of a mapped property. In some cases, the data we want the user to see will differ from the raw value in the database. In this example, pictures are stored as the path relative to the pictures folder such as '16/2_649182f5c5216.jpg'. To a user of the API, this is useless. However, we can use that data to convert it to the front-end URL needed to access that picture such as '/front/document.send.php?file=_pictures/16/2_649182f5c5216.jpg'. To accomplish this, mapped properties have the 'x-mapped-from' and 'x-mapper' fields. 'x-mapped-from' indicates the property we are mapping from. In this case, it maps from itself. 'x-mapper' is a callable that transforms the raw value to the display value. The mapper used here takes the relative path and converts it to the front-end URL. It then handles returning the default user picture if it cannot get the user's specific picture.

Partial vs Full Schema

A full schema is the defacto representation of an item in the API. In some cases, we do not want every property for an item to be visible such as dropdown types related to a main item. In `Computer` item we may show the ID and name of the computer's location, but the Location type itself has additional data like geolocation coordinates. The partial schema contains only the properties needed for the user to know where to look for the full details and some basic information about it.

Joins

Schemas may include data from tables other than the table for the main item. Most of the item, joins are used in 'object' type properties such as when bringing in an ID and name for a dropdown type. In some cases though, joins may be defined on scalar properties (not array or object).

The information required to join data from outside of the main item's table is defined inside of an 'x-join' array. The supported properties of the 'x-join' definition are:

- table: The database table to pull the data from
- fkey: The SQL field in the main table to use to identify which records in the other table are related
- field: The SQL field in the other table to match against the fkey.
- primary-property: Optional property which indicates the primary property of the foreign data. Typically, this is the 'id' field. By default, the API will assume the field specified in 'field' is the primary property. If it isn't, it is required to specify it here. In the User schema example, email addresses have a many-to-one relation with users. So, we use the user's ID field and match it against the 'users_id' field of the email addresses. In that case, the

'field' is 'users_id' but the primary property is 'id', so we need to hint to the API that 'id' is still the primary property.

- ref-join: In some cases, there is no direct connection between the main item's table and the table with the data desired (typically seen with many-to-many relations). In that case, a reference or in-between join can be specified. The 'ref_join' property follows the same format as 'x-join' except that you cannot have another 'ref_join'.

Extension Properties

Below is a complete list of supported extension fields/properties used in OpenAPI schemas.

3.6.2 Search

As the High-Level API is decoupled from the PHP classes and search options system, a new search engine was developed to handle interacting with the database. This new search engine exists in the `\Glpi\Api\HL\Search` class. For simplicity, the search engine class also provides static methods to perform item creation, update and deletion in addition to the search/get actions.

These entrypoint methods are:

- getOneBySchema
- searchBySchema
- createBySchema
- updateBySchema
- deleteBySchema

See the PHPDoc for each method for more information.

While the standard search engine constructs a single database query to retreive item(s), the High-Level API takes multiple distinct steps and multiple queries to fetch and assemble the data given the potential complexity of schemas while keeping the schemas themselves relatively simple.

The steps are:

1. Initializing a new search. This step consists of making a new instance of the `\Glpi\Api\HL\Search` class, generating a flattened array of properties (flattens properties where the keys are the full property name in dot notation to make lookups easier) in the schema and identifying joins.
2. Construct a request to get the 'dehydrated' result. In this context, that means a result without all of the desired data. It only contains the identification data (the main item ID(s) and the IDs of joined records) and the scalar join values. Each dehydrated result is an array where the keys are the primary ID field and any full join property name. The '.' in the names are replaced with 0x1F characters (Unit separator character) to avoid confusion about what is a table/field identifier. In the case that a join property is for an array of items, the IDs are separated by a 0x1D character (Group separator character). If there are no results for a specific join, a null byte character will be used. The reason a dehydrated result is fetched first is that we don't need to either worry about grouping data or handling the multiple rows returned that relate to a single main item.
3. Hydrate each of the dehydrated results. In separate queries, the search engine will fetch the data for the main item and each join. Each time a new record is fetched, it is stored in a separate array that acts like a cache to avoid fetching the same record twice.
4. Assemble the hydrated records into the final result(s). The search engine enumerates each property in the dehydrated result starting with the main item's ID and maps the hydrated data into a result that matches the expected schema.

5. Fixup the assembled records. Some post-processing is done after the record is fully assembled to clean some of the artifacts from the assembly process such as removing the keys for array type properties and replacing empty array values for object type properties with null.
6. Returning the result(s).

3.6.3 Versioning

The High-Level API will actively filter the routes and schema definitions based on the API version requested by the user (or default to the latest API version). The version being used is stored by the router in a *GLPI-API-Version* header in the request after being normalized based on version pinning rules (See the getting started documentation for the High-Level API). Controllers that extend *GlpiApiHLControllerAbstractController* can pass the request to the *getAPIVersion* helper function to get the API version.

Route Versions

All routes must have a *GlpiApiHLRouteVersion* attribute present. This attribute allows specifying an introduction, deprecated, and removal version. The introduction version is required.

When the router attempts to match a request to a route, it will take the versions specified on each route into account. So if a user requests API version 3, routes introduced in v4 will not be considered. Additionally, routes removed in v3 will also not be considered. Deprecation versions do not affect the route matching logic.

Schema Versions

All schemas must have a *x-version-introduced* property present. They may also have *x-version-deprecated* and *x-version-removed* properties if applicable. Individual properties within schemas may declare these version properties as well, but will use the versions from the schema itself if not.

When schemas are requested from each controller, they will be filtered based on the API version requested by the user (or default to the latest API version). If the versions on a schema make it inapplicable to the requested version, it is not returned at all from the controller. If the schema itself is applicable, each property is evaluated and inapplicable properties are removed.

3.7 Massive Actions

3.7.1 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.

3.7.2 Stages

Processing is splitted in three stages (each handled by a different file). They are determined by the `MassiveAction` constructor `$stage` parameter that determines its behaviour.

Stage 1: initial

File: `ajax/massiveaction.php`

When: The user checks items and clicks the bulk actions button.

What this stage does:

- Collects checked items (`$_POST['item'][itemtype][id] = 1`)
- Calls `MassiveAction::getAllMassiveActions()` for each itemtype → aggregates available actions
- Stores items in `$POST['items']` and the action list in `$POST['actions']`
- Displays a dropdown listing available actions
- Each change in the dropdown triggers an AJAX call to the `specialize` stage

Stage 2: specialize

File: `ajax/dropdownMassiveAction.php`

When: The user selects an action from the dropdown.

What this stage does:

- Retrieves the chosen action and its label from `$POST['actions']`

- Filters out items that do not support the action (via `action_filter` and `getForbiddenStandardMassiveAction()`)
- Extracts the processor: if the action key contains `ClassName:action_name`, the processor is `ClassName`; otherwise `MassiveAction` is used by default
- Calls `$processor::showMassiveActionsSubForm($ma)` → displays fields specific to the action
- Hidden fields are injected via `$ma->addHiddenFields()`

The processor is the class that contains the subform and the processing logic. It is encoded in the action key:

```php
<?php

// Action key with explicit processor
$actions['MyClass:my_action'] = 'My action';

// Implicit processor = MassiveAction
$actions['MassiveAction:delete'] = 'Move to trash';
```

Stage 3: process

File: `front/massiveaction.php`

When: The user submits the form from the `specialize` stage.

What this stage does:

- Initialises result counters: `ok`, `ko`, `noright`, `noaction`, `messages`
- Calls `$ma->process()` → `processForSeveralItemtypes()`
- For each remaining itemtype, calls `$processor::processMassiveActionsForOneItemtype($ma, $item, $ids)`
- Displays a progress bar
- If processing takes more than 5 seconds, reloads the page with the session identifier to continue (anti-timeout), via `$ma->itemDone()`
- Redirects to the previous page with a result message

3.7.3 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
<?php

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

3.7.4 Specific massive actions

If default massive actions are not sufficient for your needs, you can define your own massive actions. 3 methods must be defined to achieve this.

1. declare the actions in `getSpecificMassiveActions`
2. display the form in `showMassiveActionsSubForm`
3. process in `processMassiveActionsForOneItemtype`

```
<?php

...

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

     if (Session::haveRight(self::$rightname, UPDATE)) {
         $actions[self::class . MassiveAction::CLASS_ACTION_SEPARATOR . 'update_
↪visibility']
             = __('Visibility');
     }

     return $actions;
 }
```

Next, implement `showMassiveActionsSubForm` to display the form :

```
<?php

...

public 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, for processing implement `processMassiveActionsForOneItemtype` method:

```
<?php

...

static function processMassiveActionsForOneItemtype(MassiveAction $ma, CommonDBTM $item,
                                                    array $ids) {
   switch ($ma->getAction()) {
```

(continues on next page)

(continued from previous page)

```
      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`

3.8 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,
- **ruleright**: authorizations assignment rules,
- **rulesoftwarecategory**: rules for assigning a category to software,
- **ruleticket**: business rules for ticket.

Plugin could add their own set of rules.

3.8.1 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.

3.8.2 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::getCriterias()` 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.

3.8.3 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
<?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');
    }

    // return an array of criteria
    // default type can be found under Rule::getCriteriaDisplayPattern
    function getCriterias() {
        $criterias = [
            '_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',
            ],

            ...
```

(continues on next page)

(continued from previous page)

```
        ];

        return $criterias;
    }

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

            ...

        ];

        return $actions;
    }
}
```

A separator can be added in the criteria or actions lists by adding an entry with text contents. It render the following criteria/actions in an HTML fieldset with the provided text as legend.

```
<?php

class RuleMytype extends Rule {

    function getCriterias() {
        return [
            '_users_id_requester' => [...],
            'separator' => __('Additional criteria'), // can be any string, not only
↪'separator'
            '_users_id_observer' => [...],
        ];
    }

    function getActions() {
        return [
            'separator' => __('Observers'),
            '_users_id_observer' => [...],
        ];
    }
```

- src/RuleMytypeCollection.php

```
<?php

class RuleMytypeCollection extends RuleCollection {
    // a rule collection can process all rules for the input or stop
```

(continues on next page)

(continued from previous page)

```
    //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 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]
        );

        ...

    }
```

3.8.4 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 = [];
```

(continues on next page)

(continued from previous page)

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

3.8.5 Test for rule collection

Changed in version 11.0.5: plugin and core RuleCollection can change the test path by overriding the `RuleCollection::getRulesTestURL` function.

For plugins, there is currently no GenericController so you must implement it.

Here is the minimal setup:

```
<?php

namespace GlpiPlugin\MyPlugin\Controller;

use Glpi\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Attribute\Route;

final class RuleTestController extends AbstractController
{
    #[Route(
        "/rules/test", // /front/rulesengine.test.php for version previous to 11.0.5
        name: "rule_myplugin_test",
        methods: ["GET"],
    )]
    public function __invoke(Request $request): Response
    {
        // No generic RuleTestController controller for now
        include(GLPI_ROOT . "/front/rulesengine.test.php");
        return new Response();
    }
}
```

3.8.6 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, `RuleDictionnaryComputerModel` alters `model` field of `glpi_computers`.

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

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

```
<?php
```

(continues on next page)

(continued from previous page)

```
...

$CFG_GLPI["dictionnary_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
                                       );
```

3.8.7 Example: Adding Custom Actions with Customized display

In this example, we will add a `_send_message` action that:

1. Uses a **textarea** for input (instead of a standard text field).
2. Forces the action type to **"Send"**.
3. Limits input to 255 characters.

1. Define the Action

In your Rule class (e.g., `TicketRule`), override the `getActions()` method. Define your custom action and use `force_actions` to associate it with a specific action operator (like `send`).

```
<?php

...

public function getActions()
{
    $actions = parent::getActions();

    $actions['_send_message'] = [
        'type'          => 'textarea', // Custom type we will handle manually
        'name'          => __('Send a short text message', 'myplugin'),
        'force_actions' => ['send'],   // Force the 'Send' action operator; name can be␣
↪found in RuleAction::getActions
    ];

    return $actions;
}
```

2. Customize the Display

Override `displayAdditionalRuleAction()` to render your custom input field. This method allows you to output raw HTML (or use GLPI helpers) when your custom type is detected.

```
<?php
```

(continues on next page)

(continued from previous page)

```
...

#[Override]
public function displayAdditionalRuleAction(array $action, $value = '')
{
    if ($action['type'] === 'textarea') {
        // Render a textarea with a character limit
        echo "<textarea class='form-control' name='value' rows='4' maxlength='255'>" .␣
↪htmlescape($value) . "</textarea>";
        // Use the following if you don't need to limit the field maxlength
        // Html::textarea(['name' => 'value', 'value' => $value, 'display' => true, 'rows' =>␣
↪4]);
        return true;
    }
    return false;
}
```

3. Handle the Execution Logic

By default, GLPI might not handle your custom field or the "Send" action type for assignment. Override `executeActions()` to manually handle the value.

```
<?php

...

#[Override]
public function executeActions($output, $params, array $input = [])
{
    if (count($this->actions)) {
        foreach ($this->actions as $action) {
            // Intercept our specific field and action type
            if ($action->fields["field"] == '_send_message' && $action->fields["action_
↪type"] == 'send') {
                // Manually assign the value to the output
                $output[$action->fields["field"]] = $action->fields["value"];
            }
        }
    }
    return parent::executeActions($output, $params, $input);
}
```


3.9 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.

3.9.1 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
<?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!

3.9.2 Javascript Functions

Added 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'));
```


3.10 Right Management

3.10.1 Goals

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

3.10.2 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)

3.10.3 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 extends the possible values of rights, you need to declare these part 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
↪'));
```

(continues on next page)

(continued from previous page)

```
      $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*

3.10.4 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
<?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_profilerights`
      ON (`glpi_profilerights`.`profiles_id` = `glpi_profiles`.`id`)
   WHERE `glpi_profilerights`.`name` = 'ticket'
      AND `glpi_profilerights`.`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.

3.10.5 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.

3.11 Automatic actions

3.11.1 Goals

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

3.11.2 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.

3.11.3 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'));
      }
```

(continues on next page)

(continued from previous page)

```
      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"));

      $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.

3.11.4 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`.

3.12 Logging Systems

GLPI has distinct logging systems that must not be confused:

3.12.1 PHP logs

It reports code errors. It can be used for debugging. It runs on Monolog (PSR-3 standard).

Output: `files/_log/php-errors.log` and `files/_log/access-errors.log`

There are 3 handlers:

- `src/Glpi/Log/ErrorLogHandler.php` which outputs PHP errors and exceptions in `files/_log/php-errors.log`.
- `src/Glpi/Log/AccessLogHandler.php` which outputs `files/_log/access-errors.log`: HTTP access errors (4xx).
- `tests/src/Log/TestHandler.php` used only in automated testing context, contents are in memory only.

Note

Notice that other utilities also write contents in *php-errors.log* (CronTask::launch(), MailCollector::collect() and Toolbox::backtrace() for example)

Usage

Developers can read or empty the logs using regular system utilities (tail, cat, ...). Throwing an uncaught exception will write contents of the php-errors.log file. What is logged depends on configuration (see below).

Configuration

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:

The extra `config/local_define.php` file allow to change configuration.

```
<?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.

By default, level is defined to *Warning* (see *GlpiApplicationSystemConfigurator::computeConstants()*). In testing and development environments it's defined to *LogLevel::DEBUG* (see *GlpiApplicationEnvironment::getConstantsOverride()*).

3.12.2 Event Log

These logs are intended for GLPI administrators and can be viewed via the UI (Administration > Event logs). For example it can log when a ticket is created.

- File: `src/Glpi/Event.php`
- Method: `Event::log($items_id, $type, $level, $service, $event)`
- Output: `glpi_events` database table
- View: in *administration* > *Logs : Event logs* (at the very top, do not confuse with the file `event.log` below)

Usage Example

```
// Log a successful login (level 3 = Important)
Event::log(
    $user_id,
    "users",
    3,
    "login",
    sprintf(__('%1$s log in from IP %2$s'), $login, $ip)
);
```

Levels (`$level`) - GLPI Internal Scale (1-5)

- **1 — Critical**: Critical security errors. Examples: Login failure.
- **2 — Severe**: Severe errors (currently unused).
- **3 — Important**: Important events. Examples: Successful logins.
- **4 — Notices (default)**: Standard events. Examples: Add, delete, tracking.
- **5 — Complete**: All events. Full details.

Configuration

It can be changed in *administration : Setup > General : Log Level*. Global variable `$CFG_GLPI["event_loglevel"]` is then changed.

Behavior

- Events are recorded if `$level < $CFG_GLPI["event_loglevel"]`.
- Events with level 3 (more critical) are also written to `files/_log/event.log` via `Toolbox::logInFile()` (see File logs below).
- `CommonDBTM::getLogDefaultLevel()` returns the default level (4) for a class.

3.12.3 File logs - Toolbox::logInFile()

Toolbox::logInFile() is a basic file Logging utility.

Direct file writing without level handling. Used for specific logs (cron, mail, ldap, etc.). There is no level handling.

- File: `src/Toolbox.php`
- Method: `Toolbox::logInFile($name, $text, $force = false)` — @todo documenter les autres méthodes.
- Output: `files/_log/`: `event.log`, `mail.log`, `cron.log`, `ldap.log`, specified file name, etc.

Usage

```
// Log to files/_log/cron.log
Toolbox::logInFile("cron", "Task executed successfully\n");
```

Configuration

Located at *administration : Setup > General : Logs in files (SQL, email, automatic action...)*, the `switch` allows to enable/disable this feature. It can be forced using the `force` parameter of the method.

Other Toolbox methods

- `Toolbox::logDebug()` : it uses Php logger to log a php backtrace in `php-errors.log` file (same file as php logs (see PHP logs above)) with PSR level `LogLevel::DEBUG`.

```
try {
    doSomethingThatMayTriggerAnException();
} catch (Exception $e) {
    Toolbox::logDebug("Something wrong happened : " . $e->getMessage());
}
```

- `Toolbox::logInfo()` : currently not used in glpi core, same as logDebug() with `LogLevel::INFO`.

- `Toolbox::backtrace()` : may be deprecated soon, avoid using it, logs the php backtrace (or request filename) in a file (`php-errors` by default, same file as php logs (see PHP logs above)).

3.12.4 Pitfalls & Tips

- *Toolbox::logInFile()* may log nothing if disabled in configuration.
- Event::log() (business logs) : Events with level 3 may also written to `files/_log/event.log` via `Toolbox::logInFile()` (if not disabled by configuration).
- **Inverted Level Scales**
  - `Event::log()`: Lower level = more critical (1 = Critical)
  - `Monolog`: Higher level = more critical (600 = Emergency)
- Files written in phpunit tests are written in `tests/files/_log` (and playright tests/e2e/files).

3.13 Tools

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

3.13.1 locale/

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

- Translations can be compiled using *./bin/console locales:compile*
- *vendor/bin/extract-locales* is used to extract translated string to the POT file (before sending it to Transifex)

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

3.13.2 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 an Javascript files,
- …

```
$ ./tools/make_release.sh -y . mytag
# file created in /tmp/glpi-mytag.tgz
```

3.13.3 Modify and check code files headers

Update copyright header based on the contents of the `./tools/HEADER` file.

```
$ ./vendor/bin/licence-headers-check --fix
```

3.13.4 getsearchoptions.php

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

For example :

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

3.13.5 Not yet documented...

> **Note**
>
> Following scripts are not yet documented and probably broken. Feel free to open a pull request to add them!

- fk_generate.php
- ldap-glpi.ldif: An LDAP export
- testmail.php
- update_registered_ids.php: This script seems to update the Registered PCI and USB IDs

3.14 Javascript

3.14.1 Vue.js

Starting in GLPI 11.0, we have added support for Vue. .. note:

```
Only SFCs (Single-file Components) using the Components API is supported. Do not use the␣
↪Options API.
```

To ease integration, there is no Vue app mounted on the page body itself. Instead, each specific feature that uses Vue such as the debug toolbar mounts its own Vue app on a container element. Components must all be located in the `js/src/vue` folder for them to be built. Components should be grouped into subfolders as a sort of namespace separation. There are some helpers stored in the `window.Vue` global to help manage components and mount apps.

Building

Two npm commands exist which can be used to build or watch (auto-build when the sources change) the Vue components.

```
npm run build:vue
```

```
npm run watch:vue
```

The `npm run build` command will also build the Vue components in addition to the regular JS bundles.

To improve performance, the components are not built into a single file. Instead, webpack chunking is utilized. This results a single smaller entrypoint `app.js` being generated and a separate file for each component. The components that are automatically built utilize `defineAsyncComponent` to enable the loading of those components on demand.

Further optimizations can be done by directly including a Vue component inside a main component to ensure it is built into the main component's chunk to reduce the number of requests. This could be useful if the component wouldn't be

reused elsewhere. Just note that the child component would also have its own chunk generated since there is no way to exclude it.

Mounting

The Vue *createApp* function can be located at *window.Vue.createApp*. Each automatically built component is automatically tracked in *window.Vue.components*.

To create an app and mount a component, you can use the following code:

```
const app = window.Vue.createApp(window.Vue.components['Debug/Toolbar'].component);
app.mount('#my-app-wrapper');
```

Replace `Debug/Toolbar` with the relative path to your component without the `.vue` extension and `#my-app-wrapper` with an ID selector for the wrapper element which would need to already existing in the DOM.

For more information about Vue, please refer to the official documentation.

3.15 Extra

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

3.15.1 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 sent 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');
```

CHAPTER

FOUR

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!

CHAPTER

FIVE

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* and/or *src*
    - ...
  - *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* is the legacy way to host all classes,
- *src* is the new way to host classes; relying on *PSR-4 autoload*,
- 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.

PSR-4 autoload

Added in version 10.0.

In order to use the Composer PSR-4 autoloader in your plugin, must place your PHP class files in the */src* directory instead of */inc*. In this scenario the */inc* directory should no longer be present in the plugin folder structure.

The convention to be used is (Case sensitive): *namespace GlpiPluginMyplugin;*. The namespace should be added to every class in the */src* directory and per the PSR-12 PHP convention be placed in the top of your class. Classes using the *GlpiPluginMyplugin`namespaces will be loaded from: `GLPI_ROOTpluginsmypluginsrc`. To include folders inside the `/src* directory simply add them to your namespace and use keywords i.e. `namespace GlpiPluginMypluginSubFolder` will load from `GLPI_ROOTpluginsmypluginsrcSubFolder`.

Directive	Composer mapping
GlpiPlugin	maps (virtually) to /plugins or /marketplace
MyPlugin	maps to: /myplugin/src converted strtolower
SubFolder	maps to /src/SubFolder/ using provided case
ClassName	maps to ../ClassName.php using provided case apending .php

```
GLPI_ROOT/marketplace/myplugin/src/Test.php
```

```php
<?php

  namespace GlpiPlugin\MyPlugin;

  class Test extends CommonDBTM
  {
    \\ Your class code...
  }

?>
```

GLPI_ROOT/marketplace/myplugin/src/ChildClass/ResultOutcomes.php

```php
<?php

  namespace GlpiPlugin\MyPlugin\ChildClass;

  class ResultOutcomes extends CommonDBTM
  {
    \\ Your class code...
  }

?>
```

GLPI_ROOT/marketplace/myplugin/setup.php

```php
<?php

use GlpiPlugin\MyPlugin\Test;
use GlpiPlugin\Myplugin\ChildClass\ResultOutcomes;

function usingTest() : void
{
  $t = new Test();
  $r = new ResultOutcomes();
}

?>
```

Where to write files?

> **Warning**
>
> Plugins may 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');

/**
 * Hook executed during the GLPI boot sequence, before the session is actually loaded
 * and before the initialization of the active plugins.
 */
function plugin_myexample_boot() {
   // Indicates to GLPI that the `/plugins/myexample/api.php` path is stateless and␣
↪therefore
   // should not use session cookies nor check for a valid session.
   \Glpi\Http\SessionManager::registerPluginStatelessPath('myexample', '#^/api\.php#');
}
```

(continues on next page)

(continued from previous page)

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

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

/**
 * 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
```

(continues on next page)

(continued from previous page)

```
      return true;
   }

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

/**
 * Optional: defines plugin options.
 *
 * @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 wile, 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'
         ],
```

(continues on next page)

(continued from previous page)

```
         '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
<?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`
: Added 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
 *
 * @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 do 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
<?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
<?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_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
<?php
/**
 * Uninstall hook
 *
 * @return boolean
 */
function plugin_myexample_uninstall() {
   global $DB;

   $tables = [
      'configs'
   ];

   foreach ($tables as $table) {
      $tablename = 'glpi_plugin_myexample_' . $table;
      //Create table only if it does not exists yet!
      if ($DB->tableExists($tablename)) {
         $DB->queryOrDie(
            "DROP TABLE `$tablename`",
            $DB->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 `GlpiPlugin\Myexampleplugin` namespace. Note that the plugin name part of the namespace must be lowercase with the exception of the first letter. Child namespaces of `GlpiPlugin\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 `GlpiPlugin\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
<?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::autocompletionTextField(
         $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 you `hook.php` file:

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

(continues on next page)

(continued from previous page)

```
   //            [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`, `pre_itil_info_section`, `post_itil_info_section`, `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`
Added 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 unsecure).

`secured_configs`
Added 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 unsecure).

`add_javascript`
Add javascript in **all** pages headers

Added 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

Added 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`

add_javascript_anonymous_page
Add javascript in **all anonymous** pages headers

Added in version 10.0.18: 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_anonymous.js` file must be `plugin_anonymous.min.js`

add_javascript_module_anonymous_page
Add javascript module in **all anonymous** pages headers

Added in version 10.0.18: 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 `mymodule_anonymous.js` file must be `mymodule_anonymous.min.js`

add_css_anonymous_page
Add CSS stylesheet on **all anonymous** pages headers

Added in version 10.0.18: 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_anonymous.css` file must be `plugin_anonymous.min.css`

add_header_tag_anonymous_page
Add header tags in **all anonymous** pages headers

Added in version 10.0.18.

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
Added 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
Added 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' => [
```

(continues on next page)

(continued from previous page)

```
            '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
<?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
Added 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
Added in version 9.2.

Permit to filter search results.

Items display related

Hooks that permits to add display on items.

pre_itil_info_section
Added in version 11.

Before displaying ITIL object sections (Ticket, Change, Problem) Waits for a `<section>`.

post_itil_info_section
Added in version 11.

After displaying ITIL object sections (ticket, Change, Problem) Waits for a `<section>`.

pre_item_form
Added 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
Added 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
Added in version 9.2.1.

Add display from statistics tab of a item like ticket

timeline_actions
Added 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`
Added in version 10.0.0.

Display new actions in the ITIL object's answer dropdown

`show_in_timeline`
Added 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`
Added 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
<?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
<?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
<?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
<?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 retrun an array of rules.

executeActions
Actions to execute for rule. Will receive an array with `output`, `params` ans `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
<?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
<?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 Controllers

Plugin controllers follow the same principles as *core controllers* with a few differences specific to the plugin system.

> **Note**
>
> Controllers require GLPI >= 11.0.

5.6.1 Creating a plugin controller

Requirements:

- The controller file must be placed in the `src/Controller/` folder of the plugin.
- The namespace must follow PSR-4: `GlpiPlugin\MyPlugin\Controller\`.
- The controller must extend `Glpi\Controller\AbstractController` or implement the `Glpi\\DependencyInjection\\PublicService` interface.
- The controller must define a route using the `Route` attribute.
- The controller must return a `Symfony\\Component\\HttpFoundation\\Response` instance.

Controllers placed in `src/Controller/` are **automatically discovered**, no manual registration is needed.

Example for a plugin named `myplugin`:

```php
# plugins/myplugin/src/Controller/HelloController.php
<?php

namespace GlpiPlugin\Myplugin\Controller;

use Glpi\Controller\AbstractController;
use Symfony\Component\HttpFoundation\JsonResponse;
use Symfony\Component\HttpFoundation\Request;
```

(continues on next page)

(continued from previous page)

```
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Attribute\Route;

final class HelloController extends AbstractController
{
    #[Route("/Hello", name: "myplugin_hello", methods: "GET")]
    public function __invoke(Request $request): Response
    {
        return new JsonResponse(['message' => 'Hello from myplugin!']);
    }
}
```

5.6.2 URL routing

Plugin routes are automatically prefixed with the plugin base path. A route defined as `/Hello` in the `myplugin` plugin will be accessible at:

- `/plugins/myplugin/Hello` (standard plugins directory)

You do not need to include the plugin prefix in the `Route` attribute.

5.6.3 Rendering Twig templates

To render a Twig template from a plugin controller, use the `@plugin_key` prefix:

```
<?php
return $this->render('@myplugin/path/to/template.html.twig', [
    'my_variable' => $value,
]);
```

This will resolve to `plugins/myplugin/templates/path/to/template.html.twig`.

5.6.4 HTTP method constraints compatibility

> **Warning**
>
> A bug in GLPI prior to **11.0.7** caused plugin routes with method constraints other than `GET` to never match. The router context was always evaluated as `GET`, so any route declared with only `POST`, `PUT`, `DELETE`, `PATCH`, etc. would never be found.
>
> This bug was fixed in GLPI 11.0.7. If your plugin needs to support GLPI < 11.0.7, use the following workaround: include `GET` alongside the intended methods and check the actual method manually inside the controller.

Workaround for GLPI < 11.0.7:

```
<?php
//  Non-GET method only, broken on GLPI < 11.0.7
#[Route("/MyAction", name: "myplugin_my_action", methods: ['POST'])]

//  Works on all versions >= 11.0 (check the method manually if needed)
#[Route("/MyAction", name: "myplugin_my_action", methods: ['GET', 'POST'])]
public function __invoke(Request $request): Response
```

(continues on next page)

(continued from previous page)

```
{
    if (!$request->isMethod('POST')) {
        throw new \Symfony\Component\HttpKernel\Exception\MethodNotAllowedHttpException([
↪'POST']);
    }
    // ...
}
```

On GLPI >= 11.0.7, you can safely restrict routes to any HTTP method without the workaround.

5.6.5 Unauthenticated access

GLPI offers two distinct mechanisms for routes that must be accessible without a logged-in user. Choosing the right one depends on whether the route needs a session at all.

Session based: No auth check

The session is started normally (the session cookie is read and written), but no authentication check is performed.

The controller can read the current user's session if one happens to be active, but the request is also accepted from anonymous visitors.

Use this for public web pages (e.g. a public form or a login endpoint).

```
<?php
#[Route("/MyAction", name: "myplugin_my_action", methods: ['GET'])]
#[Glpi\Security\Attribute\SecurityStrategy(Glpi\Http\Firewall::STRATEGY_NO_CHECK)]
public function __invoke(Request $request): Response
{
    // Session may or may not be active so do not assume the user is logged in.
}
```

No session: Stateless

No session is started and no session cookie is sent or read. The request is fully stateless.

Use this when the controller manages its own authentication (e.g. an API endpoint that expects a token in a header).

Register the path pattern in the `plugin_{key}_init()` or `plugin_{key}_boot()` function in `setup.php`:

```
# plugins/myplugin/setup.php
<?php

function plugin_myplugin_init(): void
{
    \Glpi\Http\SessionManager::registerPluginStatelessPath('myplugin', '#^/MyApiEndpoint$
↪#');
}
```

The pattern is a regex matched against the path relative to the plugin base URL (i.e. without the `/plugins/myplugin` prefix).

5.7 Automatic actions

5.7.1 Goals

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

5.7.2 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*.

5.7.3 Register an automatic action

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

5.7.4 Unregister a task

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

5.8 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) {
   $actions = [];
   switch ($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 int the definition, we can use the `showMassiveActionsSubForm` and `processMassiveActionsForOneItemtype` in the same way as *core documentation for massive actions*:

```php
<?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"));
               }
            }
            return;

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


5.9 Tips & tricks

5.9.1 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(
   GlpiPlugin\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():
         //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) {
   //...
```

(continues on next page)

(continued from previous page)

```
}

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.

5.9.2 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
**/
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;
   }
```

(continues on next page)

(continued from previous page)

```
   return true;
}
```

5.9.3 Add several tabs

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

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

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;
}
```

5.9.4 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)];
}
```


5.10 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 extends 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.

5.10.1 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 the the provided (but not yet used) modes from the core. If you need a mode that does not exists, you can of course create yours!

In order to make you 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() {
   Config::setConfigurationValues('core', ['notifications_sms' => 0]);
   return true;
}

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

5.10.2 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' => '',
```

(continues on next page)

(continued from previous page)

```
      '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() {
      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'>";
```

(continues on next page)

(continued from previous page)

```
      $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/notificationsmssetting.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();
```

5.10.3 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`.

En 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];
```

(continues on next page)

(continued from previous page)

```
                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',
         '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!');
   }
}
```

5.10.4 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
<?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',
         '_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'];
```

(continues on next page)

(continued from previous page)

```
      $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;
   }
}
```


5.11 Unit Testing

5.11.1 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.

5.11.2 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');
```

(continues on next page)

(continued from previous page)

```
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.

5.11.3 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`.

5.11.4 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.

5.11.5 Real examples

The following plugins are a good example of how to implement Atoum tests:

- JAMF Plugin for GLPI
- Fields Plugin for GLPI

5.11.6 Further reading

The Atoum documentation is a good place to start if you are not familiar with unit testing or Atoum.

5.12 Plugin development tutorial

This tutorial explores the basic concepts of GLPI while building a simple plugin. It has been written to be explained during a training session, but most of this document could be read and used by people wanting to write plugins. Don’t hesitate to suggest enhancements or contribute at this address: https://github.com/glpi-project/docdev

Warning

Several prerequisites are required in order to follow this tutorial:

- A base knowledge of GLPI usage

- A correct level in web development:
  - PHP
  - HTML
  - CSS
  - SQL
  - JavaScript (JQuery)
- Being familiar with command line usage

In this first part, we will create a plugin named "My plugin" (key: `myplugin`). We will cover project startup as well as the setup of base elements.

5.12.1 Prerequisites

Here are all the things you need to start your GLPI plugin project:

- a functional web server,
- latest GLPI stable release installed locally
- a text editor or any IDE (like vscode or phpstorm),
- git version management software.

You may also need:

- Composer PHP dependency software, to handle PHP libraries specific for your plugin.
- Npm JavaScript dependency software, to handle JavaScript libraries specific for your plugin.

5.12.2 Start your project

Warning

If you have production data in your GLPI instance, make sure you disable all notifications before beginning the development. This will prevent sending of tests messages to users present in the imported data.

First of all, a few resources:

- Empty plugin and its documentation. This plugin is a kind of skeleton for quick starting a brand new plugin.
- Example plugin. It aims to do an exhaustive usage of GLPI internal API for plugins.

My new plugin

Clone `empty` plugin repository in you GLPI `plugins` directory:

```
cd /path/to/glpi/plugins
git clone https://github.com/pluginsGLPI/empty.git
```

You can use the `plugin.sh` script in the `empty` directory to create your new plugin. You must pass it the name of your plugin and the first version number. In our example:

```
cd empty
chmod +x plugin.sh
./plugin.sh myplugin 0.0.1
```

Note

Several conditions must be respected choosing a plugin name: no space or special character is allowed.
This name will be used to declare your plugin directory, as well as methods, constants, database tables and so on.
`My-Plugin` will therefore create the `MyPlugin` directory.
Using capital characters will cause issues for some core functions.

Keep it simple!

When running the command, a new directory `myplugin` will be created at the same level as the `empty` directory (both in `/path/to/glpi/plugin` directory) as well as files and methods associated with an empty plugin skeleton.

Note

If you cloned the `empty` project outside your GLPI instance, you can define a destination directory for your new plugin:

```
./plugin.sh myplugin 0.0.1 /path/to/another/glpi/plugins/
```

Retrieving Composer dependencies

In a terminal, run the following command:

```
cd /path/to/glpi/plugins/myplugin
composer install
```

Minimal plugin structure

- `front` directory is used to store our object actions (create, read, update, delete).
- `ajax` directory is used for ajax calls.
- Your plugin own classes will be stored in the `src` directory.
- gettext translations will be stored in the `locales` directory.
- An optional `templates` directory would contain your plugin Twig template files.
- `tools` directory provides some optional scripts from the empty plugin for development and maintenance of your plugin. It is now more common to get those scripts from `vendor` and `node_modules` directories.
- `vendor` directory contains:
  - PHP libraries for your plugin,
  - helpful tools provided by `empty` model.
- `node_modules` directory contains JavaScript libraries for your plugin.

- `composer.json` files describes PHP dependencies for your project.
- `package.json` file describes JavaScript dependencies for your project.
- `myplugin.xml` file contains data description for *publishing your plugin*.
- `myplugin.png` image is often included in previous XML file as a representation for GLPI plugins catalog
- `setup.php` file is meant to *instantiate your plugin*.
- `hook.php` file *contains your plugin basic functions* (install/uninstall, hooks, etc).

minimal setup.php

After running `plugin.sh` script, there must be a `setup.php` file in your `myplugin` directory.

It contains the following code:

setup.php

```php
<?php

define('PLUGIN_MYPLUGIN_VERSION', '0.0.1');
```

An optional constant declaration for your plugin version number used later in the `plugin_version_myplugin` function.

setup.php

```php
<?php

function plugin_init_myplugin() {
   global $PLUGIN_HOOKS;

   //hooks declarations here...
}
```

This instanciation function is important, we will declare later here *Hooks* on GLPI internal API. It's systematically called on **all** GLPI pages except if the `_check_prerequisites` fails (see below).

setup.php

```php
<?php

// Minimal GLPI version, inclusive
define("PLUGIN_MYPLUGIN_MIN_GLPI_VERSION", "10.0.0");

// Maximum GLPI version, exclusive
define("PLUGIN_MYPLUGIN_MAX_GLPI_VERSION", "10.0.99");

function plugin_version_myplugin()
{
    return [
        'name'           => 'MonNouveauPlugin',
        'version'        => PLUGIN_MYPLUGIN_VERSION,
        'author'         => '<a href="http://www.teclib.com">Teclib\'</a>',
        'license'        => 'MIT',
        'homepage'       => '',
        'requirements'   => [
```

(continues on next page)

(continued from previous page)

```
            'glpi' => [
                'min' => PLUGIN_MYPLUGIN_MIN_GLPI_VERSION,
                'max' => PLUGIN_MYPLUGIN_MAX_GLPI_VERSION,
            ]
    ];
}
```

This function specifies data that will be displayed in the `Setup > Plugins` menu of GLPI as well as some minimal constraints. We reuse the constant `PLUGIN_MYPLUGIN_VERSION` declared above. You can of course change data according to your needs.

> **Note**
>
> **Choosing a license**
>
> The choice of a license is **important** and has many consequences on the future use of your developments. Depending on your preferences, you can choose a more permissive or restrictive orientation. Websites that can be of help exists, like https://choosealicense.com/.
>
> In our example, MIT license has been choose. It's a very popular choice which gives user enough liberty using your work. It just asks to keep the notice (license text) and respect the copyright. You can't be dispossessed of your work, paternity must be kept.

setup.php

```php
<?php

function plugin_myplugin_check_config($verbose = false)
{
    if (true) { // Your configuration check
        return true;
    }

    if ($verbose) {
        _e('Installed / not configured', 'myplugin');
    }

    return false;
}
```

This function is systematically called on **all** GLPI pages. It allows to automatically deactivate plugin if defined criteria are not or no longer met (returning `false`).

minimal hook.php

This file must contains installation and uninstallation functions:

hook.php

```php
<?php

function plugin_myplugin_install()
{
```

(continues on next page)

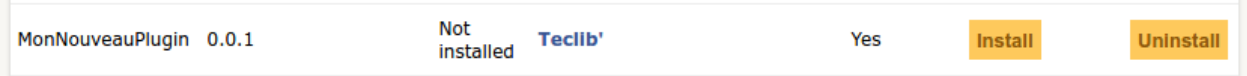
(continued from previous page)

```
    return true;
}

function plugin_myplugin_uninstall()
{
    return true;
}
```

When all steps are OK, we must return `true`. We will populate these functions later while creating/removing database tables.

Install your plugin



Following those first steps, you should be able to install and activate your plugin from `Setup` > `Plugins` menu.

5.12.3 Creating an object

In this part, we will add an itemtype to our plugin and make it interact with GLPI.
This will be a parent object that will regroup several "assets".
We will name it "Superasset".

CommonDBTM usage and classes creation

This super class adds the ability to manage items in the database. Your working classes (in the `src` directory) can inherit from it and are called "itemtype" by convention.

Note

Conventions:

- Classes must respect PSR-12 naming conventions. We maintain a *guide on coding standards*
- *SQL tables* related to your classes must respect that naming convention: `glpi_plugin_pluginkey_names`
  - a global `glpi_` prefix
  - a prefix for plugins `plugin_`
  - plugin key `myplugin_`
  - itemtype name in plural form `superassets`
- *Tables columns* must also follow some conventions:
  - there must be an `auto-incremented primary` field named `id`
  - foreign keys names use that referenced table name without the global `glpi_` prefix and with `_id` suffix. example: `plugin_myotherclasses_id` references `glpi_plugin_myotherclasses` table

  Warning! GLPI does not use database foreign keys constraints. Therefore you must not use `FOREIGN` or `CONSTRAINT` keys.
- Some extra advice:

- always end your files with an extra carriage return
- never use the closing PHP tag ?> - see https://www.php.net/manual/en/language.basic-syntax.instruction-separation.php

Main reason for that is to avoid concatenation errors when using require/include functions, and prevent unexpected outputs.

We will create our first class in `Superasset.php` file in our plugin `src` directory:

We declare a few parts:

src/Superasset.php

```php
<?php
namespace GlpiPlugin\Myplugin;

use CommonDBTM;

class Superasset extends CommonDBTM
{
    // right management, we'll change this later
    static $rightname = 'computer';

    /**
     *  Name of the itemtype
     */
    static function getTypeName($nb=0)
    {
        return _n('Super-asset', 'Super-assets', $nb);
    }
}
```

Warning

`namespace` must be CamelCase

Note

Here are most common CommonDBTM inherited methods:

add(array $input) : Add an new object in database table. `input` parameter contains table fields. If add goes well, the object will be populated with provided data. It returns the id of the new added line, or `false` if there were an error.

```php
<?php

namespace GlpiPlugin\Myplugin;

$superasset = new Superasset;
$superassets_id = $superasset->add([
    'name' => 'My super asset'
]);
if (!superassets_id) {
    //super asset has not been created :'(
}
```

getFromDB(integer $id) : load an item from database into current object using its id. Fetched data will be available from `fields` object property. It returns `false` if the object does not exists.

```php
<?php

if ($superasset->getFromDB($superassets_id)) {
    //super $superassets_id has been lodaded.
    //you can access its data from $superasset->fields
}
```

update(array $input) : update fields of `id` identified line with `$input` parameter. The `id` key must be part of `$input`. Returns a boolean.

```php
<?php

if (
    $superasset->update([
        'id'      => $superassets_id,
        'comment' => 'my comments'
    ])
) {
    //super asset comment has been updated in databse.
}
```

delete(array $input, bool $force = false) : remove `id` identified line corresponding. The `id` key must be part of `$input`. `$force` parameter indicates if the line must be place in trashbin (`false`, and a `is_deleted` field must be present in the table) or removed (`true`). Returns a boolean.

```php
<?php

if ($superasset->delete(['id' => $superassets_id])) {
    //super asset has been moved to trashbin
}

if ($superasset->delete(['id' => $superassets_id], true)) {
    //super asset is no longer present in database.
    //a message will be displayed to user on next displayed page.
}
```

Installation

In the `plugin_myplugin_install` function of your `hook.php` file, we will manage the creation of the database table corresponding to our itemtype `Superasset`.

hook.php

```php
<?php

use DBConnection;
use GlpiPlugin\Myplugin\Superasset;
use Migration;

function plugin_myplugin_install()
{
    global $DB;
```

(continues on next page)

(continued from previous page)

```php

    $default_charset   = DBConnection::getDefaultCharset();
    $default_collation = DBConnection::getDefaultCollation();

    // instantiate migration with version
    $migration = new Migration(PLUGIN_MYPLUGIN_VERSION);

    // create table only if it does not exist yet!
    $table = Superasset::getTable();
    if (!$DB->tableExists($table)) {
        //table creation query
        $query = "CREATE TABLE `$table` (
                  `id`          int unsigned NOT NULL AUTO_INCREMENT,
                  `is_deleted` TINYINT NOT NULL DEFAULT '0',
                  `name`       VARCHAR(255) NOT NULL,
                  PRIMARY KEY  (`id`)
                ) ENGINE=InnoDB
                DEFAULT CHARSET={$default_charset}
                COLLATE={$default_collation}";
        $DB->doQuery($query);
    }

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

    return true;
}
```

In addition, of a primary key, `VARCHAR` field to store a name entered by the user and a flag for the the trashbin.

> **Note**
>
> You of course can add some other fields with other types (stay reasonable).

To handle migration from a version to another of our plugin, we will use GLPI Migration class.

hook.php

```php
<?php

use Migration;

function plugin_myplugin_install()
{
    global $DB;

    // instantiate migration with version
    $migration = new Migration(PLUGIN_MYPLUGIN_VERSION);

    ...

    if ($DB->tableExists($table)) {
```

(continues on next page)

(continued from previous page)

```
        // missing field
        $migration->addField(
            $table,
            'fieldname',
            'string'
        );

        // missing index
        $migration->addKey(
            $table,
            'fieldname'
        );
    }

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

    return true;
}
```

Warning

Migration class provides several methods that permit to manipulate tables and fields. All calls will be stored in queue that will be executed when calling `executeMigration` method.

Here are some examples:

addField($table, $field, $type, $options)
: adds a new field to a table

changeField($table, $oldfield, $newfield, $type, $options)
: change a field name or type

dropField($table, $field)
: drops a field

dropTable($table)
: drops a table

renameTable($oldtable, $newtable)
: rename a table

See Migration documentation for all other possibilities.

`$type` parameter of different functions is the same as the private Migration::fieldFormat() method it allows shortcut for most common SQL types (bool, string, integer, date, datetime, text, longtext, autoincrement, char)

Uninstallation

To uninstall our plugin, we want to clean all related data.

hook.php

```
<?php

```

(continues on next page)

(continued from previous page)

```php
use GlpiPlugin\Myplugin\Superasset;

function plugin_myplugin_uninstall()
{
    global $DB;

    $tables = [
        Superasset::getTable(),
    ];

    foreach ($tables as $table) {
        if ($DB->tableExists($table)) {
            $DB->doQuery(
                "DROP TABLE `$table`"
            );
        }
    }

   return true;
}
```

Framework usage

Some useful functions

```php
<?php

Toolbox::logError($var1, $var2, ...);
```

This method stored in `glpi/files/_log/php-errors.log` file content of its parameters (may be strings, arrays, objects, etc).

```php
<?php

Html::printCleanArray($var);
```

This method will display a "debug" array of the provided variable. It only accepts `array` type.

5.12.4 Common actions on an object

Note

We will now add most common actions to our `Superasset` itemtype:

- display a list and a form to add/edit
- define add/edit/delete routes

In our `front` directory, we will need two new files.

Warning

Into those files, we will import GLPI framework with the following:

```
<?php

include ('../../../inc/includes.php');
```

First file (`superasset.php`) will display list of items stored in our table.

It will use the internal search engine `show` method of the *search engine*.

front/superasset.php

```
<?php

use GlpiPlugin\Myplugin\Superasset;
use Search;
use Html;

include ('../../../inc/includes.php');

Html::header(
    Superasset::getTypeName(),
    $_SERVER['PHP_SELF'],
    "plugins",
    Superasset::class,
    "superasset"
);
Search::show(Superasset::class);
Html::footer();
```

`header` and `footer` methods from Html class permit to rely on GLPI graphical user interface (menu, breadcrumb, page footer, etc).

Second file (`superasset.form.php` - with `.form` suffix) will handle CRUD actions.

front/superasset.form.php

```
<?php

use GlpiPlugin\Myplugin\Superasset;
use Html;

include ('../../../inc/includes.php');

$supperasset = new Superasset();

if (isset($_POST["add"])) {
    $newID = $supperasset->add($_POST);

    if ($_SESSION['glpibackcreated']) {
        Html::redirect(Superasset::getFormURL()."?id=".$newID);
    }
    Html::back();
```

(continues on next page)

(continued from previous page)

```

} else if (isset($_POST["delete"])) {
    $supperasset->delete($_POST);
    $supperasset->redirectToList();

} else if (isset($_POST["restore"])) {
    $supperasset->restore($_POST);
    $supperasset->redirectToList();

} else if (isset($_POST["purge"])) {
    $supperasset->delete($_POST, 1);
    $supperasset->redirectToList();

} else if (isset($_POST["update"])) {
    $supperasset->update($_POST);
    \Html::back();

} else {
    // fill id, if missing
    isset($_GET['id'])
        ? $ID = intval($_GET['id'])
        : $ID = 0;

    // display form
    Html::header(
       Superasset::getTypeName(),
       $_SERVER['PHP_SELF'],
       "plugins",
       Superasset::class,
       "superasset"
    );
    $supperasset->display(['id' => $ID]);
    Html::footer();
}
```

All common actions defined here are handled from CommonDBTM class. For missing display action, we will create a `showForm` method in our `Superasset` class. Note this one already exists in `CommonDBTM` and is displayed using a generic Twig template.

We will use our own template that will extends the generic one (because it only displays common fields).

src/Superasset.php

```
<?php

namespace GlpiPlugin\Myplugin;

use CommonDBTM;
use Glpi\Application\View\TemplateRenderer;

class Superasset extends CommonDBTM
{

```

(continues on next page)

(continued from previous page)

```
    ...

    function showForm($ID, $options=[])
    {
        $this->initForm($ID, $options);
        // @myplugin is a shortcut to the **templates** directory of your plugin
        TemplateRenderer::getInstance()->display('@myplugin/superasset.form.html.twig', [
            'item'   => $this,
            'params' => $options,
        ]);

        return true;
    }
}
```

templates/superasset.form.html.twig

```
{% extends "generic_show_form.html.twig" %}
{% import "components/form/fields_macros.html.twig" as fields %}

{% block more_fields %}
    blabla
{% endblock %}
```

After that step, a call in our browser to *http://glpi/plugins/myplugin/front/superasset.form.php* should display the creation form.

Warning

`components/form/fields_macros.html.twig` file imported in the example includes Twig functions or macros to display common HTML fields like:

`{{ fields.textField(name, value, label = '', options = {}) }}` : HTML code for a `text` input.

`{{ fields.hiddenField(name, value, label = '', options = {}) }` : HTML code for a `hidden` input.

`{{ fields.dateField(name, value, label = '', options = {}) }` : HTML code for a date picker (using flatpickr)

`{{ fields.datetimeField(name, value, label = '', options = {}) }` : HTML code for a datetime picker (using flatpickr)

See `templates/components/form/fields_macros.html.twig` file in source code for more details and capacities.

5.12.5 Adding to menu and breadcrumb

We would like to access our pages without entering their URL in our browser.

We'll therefore define our first *Hook* in our plugin `init`.

Open `setup.php` and edit `plugin_init_myplugin` function:

setup.php

```php
<?php

use GlpiPlugin\Myplugin\Superasset;

function plugin_init_myplugin()
{
    ...

    // add menu hook
    $PLUGIN_HOOKS[Hooks::MENU_TOADD]['myplugin'] = [
        // insert into 'plugin menu'
        'plugins' => Superasset::class
    ];
}
```

This *hook* indicates our `Superasset` itemtype defines a menu display function. Edit our class and add related methods:

src/Superasset.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use CommonDBTM;

class Superasset extends CommonDBTM
{
    ...

    /**
     * Define menu name
     */
    static function getMenuName($nb = 0)
    {
        // call class label
        return self::getTypeName($nb);
    }

    /**
     * Define additional links used in breacrumbs and sub-menu
     *
     * A default implementation is provided by CommonDBTM
     */
    static function getMenuContent()
    {
        $title  = self::getMenuName(Session::getPluralNumber());
        $search = self::getSearchURL(false);
        $form   = self::getFormURL(false);

        // define base menu
        $menu = [
            'title' => __("My plugin", 'myplugin'),
            'page'  => $search,
```

(continues on next page)

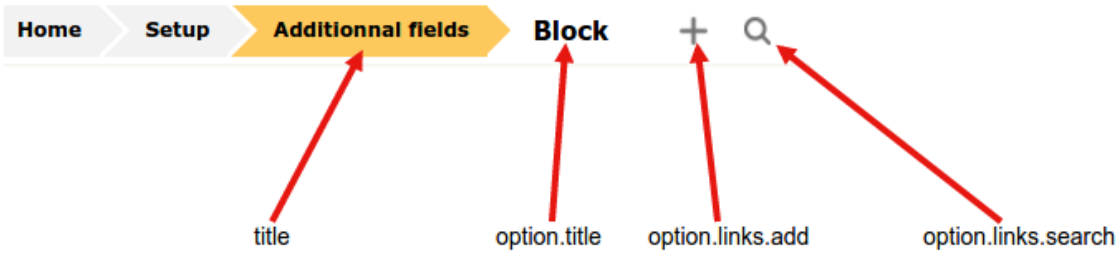
(continued from previous page)

```
            // define sub-options
            // we may have multiple pages under the "Plugin > My type" menu
            'options' => [
                'superasset' => [
                    'title' => $title,
                    'page'  => $search,

                    //define standard icons in sub-menu
                    'links' => [
                        'search' => $search,
                        'add'    => $form
                    ]
                ]
            ]
        ];

        return $menu;
    }
}
```

`getMenuContent` function may seem redundant at first, but each of the coded entries relates to different parts of the display. The `options` part is used to have a fourth level of breadcrumb and thus have a clickable submenu in your entry page.



Each `page` key is used to indicate on which URL the current part applies.

Note

GLPI menu is loaded in `$_SESSION['glpimenu']` on login. To see your changes, either use the `DEBUG` mode, or disconnect and reconnect.

Note

It is possible to have only one menu level for the plugin (3 globally), just move the `links` part to the first level of the `$menu` array.

Note

It is also possible to define custom `links`. You just need to replace the key (for example, add or search) with an html containing an image tag:

```
'links' = [
    '<img src="path/to/my.png" title="my custom link">' => $url
]
```

5.12.6 Defining tabs

GLPI proposes three methods to define tabs:

defineTabs(array $options = []): declares classes that provides tabs to current class.

getTabNameForItem(CommonGLPI $item, boolean $withtemplate = 0): declares titles displayed for tabs.

displayTabContentForItem(CommonGLPI $item, integer $tabnum = 1, boolean $withtemplate = 0): allow displaying tabs contents.

Standards tabs

Some GLPI internal API classes allows you to add a behaviour with minimal code.

It's true for notes (Notepad) and history (Log).

Here is an example for both of them:

src/Superasset.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use CommonDBTM;
use Notepad;
use Log;

class Superasset extends CommonDBTM
{
    // permits to automaticaly store logs for this itemtype
    // in glpi_logs table
    public $dohistory = true;

    ...

    function defineTabs($options = [])
    {
        $tabs = [];
        $this->addDefaultFormTab($tabs)
            ->addStandardTab(Notepad::class, $tabs, $options)
            ->addStandardTab(Log::class, $tabs, $options);

        return $tabs;
    }
}
```

Display of an instance of your itemtype from the page `front/superasset.php?id=1` should now have 3 tabs:

- Main tab with your itemtype name
- Notes tab
- History tab

Custom tabs

On a similar basis, we can target another class of our plugin:

src/Superasset.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use CommonDBTM;
use Notepad;
use Log;

class Superasset extends CommonDBTM
{
    // permits to automaticaly store logs for this itemtype
    // in glpi_logs table
    public $dohistory = true;

    ...

    function defineTabs($options = [])
    {
        $tabs = [];
        $this->addDefaultFormTab($tabs)
            ->addStandardTab(Superasset_Item::class, $tabs, $options)
            ->addStandardTab(Notepad::class, $tabs, $options)
            ->addStandardTab(Log::class, $tabs, $options);

        return $tabs;
    }
```

In this new class we will define two other methods to control title and content of the tab:

src/Superasset_Item.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use CommonDBTM;
use Glpi\Application\View\TemplateRenderer;

class Superasset_Item extends CommonDBTM
{
    /**
     * Tabs title
```

(continues on next page)

(continued from previous page)

```
     */
    function getTabNameForItem(CommonGLPI $item, $withtemplate = 0)
    {
        switch ($item->getType()) {
            case Superasset::class:
                $nb = countElementsInTable(self::getTable(),
                    [
                        'plugin_myplugin_superassets_id' => $item->getID()
                    ]
                );
                return self::createTabEntry(self::getTypeName($nb), $nb);
        }
        return '';
    }

    /**
     * Display tabs content
     */
    static function displayTabContentForItem(CommonGLPI $item, $tabnum = 1,
↪$withtemplate = 0)
    {
        switch ($item->getType()) {
            case Superasset::class:
                return self::showForSuperasset($item, $withtemplate);
        }

        return true;
    }

    /**
     * Specific function for display only items of Superasset
     */
    static function showForSuperasset(Superasset $superasset, $withtemplate = 0)
    {
        TemplateRenderer::getInstance()->display('@myplugin/superasset_item_.html.twig',␣
↪[
            'superasset' => $superasset,
        ]);
    }
}
```

As previously, we will use a Twig template to handle display.

templates/superasset_item.html.twig

```
{% import "components/form/fields_macros.html.twig" as fields %}

example content
```

> **Note**
>
> **Exercise**: For the rest of this part, you will need to complete our plugin to allow the installa-

tion/uninstallation of the data of this new class `Superasset_Item`.

Table should contains following fields:

- an identifier (id)
- a foreign key to `plugin_myplugin_superassets` table
- two fields to link with an itemtype:
  - `itemtype` which will store the itemtype class to link to (Computer for example)
  - `items_id` the id of the linked asset

Your plugin must be re-installed or updated for the table creation to be done. You can force the plugin status to change by incrementing the version number in the `setup.php` file.

For the exercise, we will only display computers (Computer) displayed with the following code:

```
{{ fields.dropdownField(
    'Computer',
    'items_id',
    '',
    __('Add a computer')
) }}
```

We will include a mini form to insert related items in our table. Form actions can be handled from `myplugin/front/supperasset.form.php` file.

Prior to GLPI 12, GLPI forms submitted as `POST` will be protected with a CRSF token.. You can include a hidden field to validate the form:

```
<input type="hidden" name="_glpi_csrf_token" value="{{ csrf_token() }}">
```

This has no effect on GLPI 12 and can be omitted. For further information, read CSRF protection.

We will also display a list of computers already associated below the form.

Using core objects

We can also allow our class to add tabs on core objects. We will declare this in a new line in our `init` function:

setup.php

```
<?php

use Computer;

function plugin_init_myplugin()
{
   ...

    Plugin::registerClass(GlpiPlugin\Myplugin\Superasset_Item::class, [
        'addtabon' => Computer::class
    ]);
}
```

Title and content for this tab are done as previously with:

- `CommonDBTM::getTabNameForItem()`
- `CommonDBTM::displayTabContentForItem()`

> **Note**
>
> **Exercise**: Complete previous methods to display on computers a new tab with associated `Superasset`.

5.12.7 Defining Search options

Search options is an array of columns for GLPI search engine. They are used to know for each itemtype how the database must be queried, and how data should be displayed.

In our class, we must declare a `rawSearchOptions` method:

src/Superasset.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use CommonDBTM;

class Superasset extends CommonDBTM
{
    ...

    function rawSearchOptions()
    {
        $options = [];

        $options[] = [
            'id'   => 'common',
            'name' => __('Characteristics')
        ];

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

        $options[] = [
            'id'    => 2,
            'table' => self::getTable(),
            'field' => 'id',
            'name'  => __('ID')
        ];

        $options[] = [
            'id'           => 3,
            'table'        => Superasset_Item::getTable(),
            'field'        => 'id',
            'name'         => __('Number of associated assets', 'myplugin'),
            'datatype'     => 'count',
```

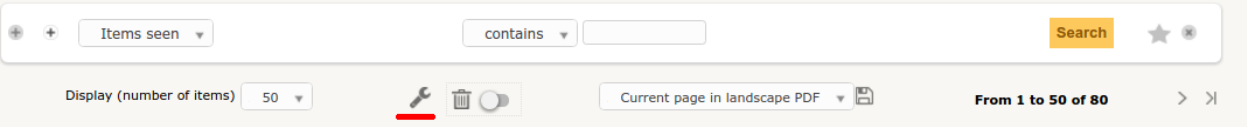
(continues on next page)

(continued from previous page)

```
            'forcegroupby' => true,
            'usehaving'    => true,
            'joinparams'   => [
                'jointype' => 'child',
            ]
        ];

        return $options;
    }
}
```

Following this addition, we should be able to select our new columns from our asset list page:



Those options will also be present in search criteria list of that page.

Each `option` is identified by an `id` key. This key is used in other parts of GLPI. It **must be absolutely unique**. By convention, '1' and '2' are "reserved" for the object name and ID.

The *search options documentation* describes all possible options.

Using other objects

It is also possible to improve another itemtype search options. As an example, we would like to display associated "Superasset" on in the computer list:

hook.php

```
<?php

use GlpiPlugin\Myplugin\Superasset;
use GlpiPlugin\Myplugin\Superasset_Item;

...

function plugin_myplugin_getAddSearchOptionsNew($itemtype)
{
    $sopt = [];

    if ($itemtype == 'Computer') {
        $sopt[] = [
            'id'           => 12345,
            'table'        => Superasset::getTable(),
            'field'        => 'name',
            'name'         => __('Associated Superassets', 'myplugin'),
            'datatype'     => 'itemlink',
            'forcegroupby' => true,
            'usehaving'    => true,
            'joinparams'   => [
                'beforejoin' => [
                    'table'      => Superasset_Item::getTable(),
```

(continues on next page)

(continued from previous page)

```
                    'joinparams' => [
                        'jointype' => 'itemtype_item',
                    ]
                ]
            ]
        ];
    }

    return $sopt;
}
```

As previously, you must provide an `id` for your new search options that does not override existing ones for `Computer`.

You can use a script from the `tools` folder of the GLPI git repository (not present in the "release" archives) to help you list the **id** already declared (by the core and plugins present on your computer) for a particular itemtype.

```
/usr/bin/php /path/to/glpi/tools/getsearchoptions.php --type=Computer
```

5.12.8 Search engine display preferences

We just have added new columns to our itemtype list. Those columns are handled by `DisplayPreference` object (`glpi_displaypreferences` table). They can be defined as global (set `0` for `users_id` field) or personal (set `users_id` field to the user id). They are sorted (`rank` field) and target an itemtype plus a `searchoption` (`num` field).

Warning

Warning Global preferences are applied to all users that don't have any personal preferences set.

Note

Exercise: You will change installation and uninstallation functions of your plugin to add and remove global preferences so objects list display some columns.

5.12.9 Standard events hooks

During a GLPI object life cycle, we can intervene via our plugin before and after each event (add, modify, delete).

For our own objects, following methods can be implemented:

- prepareInputForAdd
- post_addItem
- prepareInputForUpdate
- post_updateItem
- pre_deleteItem
- post_deleteItem
- post_purgeItem

For every event applied on the database, we have a method that is executed before, and another after.

> **Note**
>
> **Exercise**: Add required methods to `Superasset` class to check the `name` field is properly filled when adding and updating.
>
> On effective removal, we must ensure linked data from other tables are also removed.

Plugins can also intercept standard core events to apply changes (or even refuse the event). Here are the names of the *hooks*:

```php
<?php

use Glpi\Plugin\Hooks;

...

Hooks::PRE_ITEM_ADD;
Hooks::ITEM_ADD;
Hooks::PRE_ITEM_DELETE;
Hooks::ITEM_DELETE;
Hooks::PRE_ITEM_PURGE;
Hooks::ITEM_PURGE;
Hooks::PRE_ITEM_RESTORE;
Hooks::ITEM_RESTORE;
Hooks::PRE_ITEM_UPDATE;
Hooks::ITEM_UPDATE;
```

More information are available from *hooks documentation* especially on *standard events* part.

For all those calls, we will get an instance of the current object in parameter of our `callback` function. We will be able to access its current fields (`$item->fields`) or those sent by the form (`$item->input`). As all PHP objects, this instance will be passed by reference.

We will declare one of those hooks usage in the plugin init function and add a `callback` function:

setup.php

```php
<?php

use GlpiPlugin\Myplugin\Superasset;

...

function plugin_init_myplugin()
{
   ...

    // callback a function (declared in hook.php)
    $PLUGIN_HOOKS[Hooks::ITEM_UPDATE]['myplugin'] = [
        'Computer' => 'myplugin_computer_updated'
    ];

    // callback a class method
    $PLUGIN_HOOKS[Hooks::ITEM_ADD]['myplugin'] = [
         'Computer' => [
```

(continues on next page)

(continued from previous page)

```
                Superasset::class, 'computerUpdated'
            ]
        ];
}
```

In both cases (`hook.php` function or class method), the prototype of the functions will be made on this model:

```php
<?php

use CommonDBTM;
use Session;

function hookCallback(CommonDBTM $item)
{
    ...

    // if we need to stop the process (valid for pre* hooks)
    if ($mycondition) {
        // clean input
        $item->input = [];

        // store a message in session for warn user
        Session::addMessageAfterRedirect('Action forbidden because...');

        return;
   }
}
```

> **Note**
>
> **Exercise**: Use a *hook* to intercept the purge of a computer and remove associated with a `Superasset` lines if any.

5.12.10 Importing libraries (JavaScript / CSS)

Plugins can declare import of additional libraries from their `init` function.

setup.php

```php
<?php

use Glpi\Plugin\Hooks;

function plugin_init_myplugin()
{
    ...

    // css & js
    $PLUGIN_HOOKS[Hooks::ADD_CSS]['myplugin'] = 'myplugin.css';
    $PLUGIN_HOOKS[Hooks::ADD_JAVASCRIPT]['myplugin'] = [
        'js/common.js',
    ];
```

(continues on next page)

(continued from previous page)

```
    // on ticket page (in edition)
    if (strpos($_SERVER['REQUEST_URI'], "ticket.form.php") !== false
        && isset($_GET['id'])) {
        $PLUGIN_HOOKS[Hooks::ADD_JAVASCRIPT]['myplugin'][] = 'js/ticket.js.php';
    }

    ...
}
```

Several things to remember:

- Loading paths are relative to plugin *public* directory.
- Scripts declared this way will be loaded on **all** GLPI pages. You must check the current page in the `init` function.
- You can rely on `Html::requireJs()` method to load external resources. Paths will be prefixed with GLPI root URL at load.
- If you want to modify page DOM and especially what is displayed in main form, you should call your code twice (on page load and on current tab load) and add a class to check the effective application of your code:

```
$(function() {
    doStuff();
    $(".glpi_tabs").on("tabsload", function(event, ui) {
        doStuff();
    });
});

var doStuff = function()
{
    if (! $("html").hasClass("stuff-added")) {
        $("html").addClass("stuff-added");

        // do stuff you need
        ...

    }
};
```

Note

Exercises:

1. Add a new icon in preferences menu to display main GLPI configuration. You can use tabler-icons:

- `<a href='...' class='ti ti-mood-smile'></a>`

1. On ticket edition page, add an icon to self-associate as a requester on the model of the one present for the "assigned to" part.

5.12.11 Display hooks

Since GLPI 9.1.2, it is possible to display data in native objects forms via new hooks. See *display related hooks* in plugins documentation.

As previous *hooks*, declaration will look like:

setup.php

```php
<?php

use Glpi\Plugin\Hooks;
use GlpiPlugin\Myplugin\Superasset;

function plugin_init_myplugin()
{
   ...

    $PLUGIN_HOOKS[Hooks::PRE_ITEM_FORM]['myplugin'] = [
        Superasset::class, 'preItemFormComputer'
    ];
}
```

Warning

Important Those display hooks are a bit different from other hooks regarding parameters that are passed to callback underlying method. We will obtain an array with the following keys:

- `item` with current `CommonDBTM` object
- `options`, an array passed from current object `showForm()` method

example of a call from core:

```php
<?php

Plugin::doHook("pre_item_form", ['item' => $this, 'options' => &$options]);
```

Note

Exercice: Add the number of associated `Superasset` in the computer form header. It should be a link to the *previous added tab* to computers. This link will target the same page, but with the `forcetab=PluginMypluginSuperasset$1` parameter.

5.12.12 Adding a configuration page

We will add a tab in GLPI configuration so some parts of our plugin can be optional.

We previously added a tab to the form for computers using hooks in `setup.php` file. We will define two configuration options to enable/disable those tabs.

GLPI provides a `glpi_configs` table to store software configuration. It allows plugins to save their own data without defining additional tables.

First of all, let's create a new `Config.php` class in the `src/` folder with the following skeleton:

src/Config.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use CommonGLPI;
use Dropdown;
use Html;
use Session;
use Glpi\Application\View\TemplateRenderer;

class Config extends \Config
{

    static function getTypeName($nb = 0)
    {
        return __('My plugin', 'myplugin');
    }

    static function getConfig()
    {
        return \Config::getConfigurationValues('plugin:myplugin');
    }

    function getTabNameForItem(CommonGLPI $item, $withtemplate = 0)
    {
        switch ($item->getType()) {
            case \Config::class:
                return self::createTabEntry(self::getTypeName());
        }
        return '';
    }

    static function displayTabContentForItem(
        CommonGLPI $item,
        $tabnum = 1,
        $withtemplate = 0
    ) {
        switch ($item->getType()) {
            case \Config::class:
                return self::showForConfig($item, $withtemplate);
        }

```

(continues on next page)

(continued from previous page)

```php
        return true;
    }

    static function showForConfig(
        \Config $config,
        $withtemplate = 0
    ) {
        global $CFG_GLPI;

        if (!self::canView()) {
            return false;
        }

        $current_config = self::getConfig();
        $canedit        = Session::haveRight(self::$rightname, UPDATE);

        TemplateRenderer::getInstance()->display('@myplugin/config.html.twig', [
            'current_config' => $current_config,
            'can_edit'       => $canedit
        ]);
    }
}
```

Once again, we manage display from a dedicated template file:

templates/config.html.twig

```twig
{% import "components/form/fields_macros.html.twig" as fields %}

{% if can_edit %}
    <form name="form" action="{{ "Config"|itemtype_form_path }}" method="POST">
        <input type="hidden" name="config_class" value="GlpiPlugin\\Myplugin\\Config">
        <input type="hidden" name="config_context" value="plugin:myplugin">
        <input type="hidden" name="_glpi_csrf_token" value="{{ csrf_token() }}"> {# useless for GLPI 12 #}

        {{ fields.dropdownYesNo(
            'myplugin_computer_tab',
            current_config['myplugin_computer_tab'],
            __("Display tab in computer", 'myplugin')
        ) }}

        {{ fields.dropdownYesNo(
            'myplugin_computer_form',
            current_config['myplugin_computer_form'],
            __("Display information in computer form", 'myplugin')
        ) }}

        <button type="submit" class="btn btn-primary mx-1" name="update" value="1">
            <i class="ti ti-device-floppy"></i>
            <span>{{ _x('button', 'Save') }}</span>
        </button>
    </form>
```

(continues on next page)

(continued from previous page)

```
{% endif %}
```

This skeleton retrieves the calls to a tab in the `Setup > General` menu to display the dedicated form. It is useless to add a `front` file because the GLPI `Config` object already offers a form display.

Note that we display, from the `myplugin_computer_form` two yes/no fields named `myplugin_computer_tab` and `myplugin_computer_form`.

Note

Complete `setup.php` file by defining the new tab in the `Config` class.

You also have to add those new configuration entries management to install/uninstall methods. You can use the following:

```php
<?php

use Config;

Config::setConfigurationValues('##context##', [
    '##config_name##' => '##config_default_value##'
]);
```

```php
<?php

use Config;

$config = new Config();
$config->deleteByCriteria(['context' => '##context##']);
```

Do not forget to replace ## surrounded terms with your own values!

5.12.13 Managing rights

To limit access to our plugin features to some of our users, we can use the GLPI Profile class.

This will check `$rightname` property of class that inherits CommonDBTM for all standard events. Those check are done by static `can*` functions:

- canCreate for add
- canUpdate for update
- canDelete for delete
- canPurge for delete when `$force` parameter is set to `true`

In order to customize rights, we will redefine those static methods in our classes.

If we need to check a right manually in our code, the Session class provides some methods:

```php
<?php

use Session;

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

(continues on next page)

(continued from previous page)

```
}

// 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
}
```

Above methods return a boolean. If we need to stop the page with a message to the user, we can use equivalent methods with `check` instead of `have` prefix:

- checkRight
- checkRightsOr

Warning

If you need to check a right in an SQL query, use bitwise operators & and |:

```
<?php

$iterator = $DB->request([
    'SELECT' => 'glpi_profiles_users.users_id',
    'FROM' => 'glpi_profiles_users',
    'INNER JOIN' => [
        'glpi_profiles' => [
            'ON' => [
                'glpi_profiles_users' => 'profiles_id'
                 'glpi_profiles' => 'id'
            ]
        ],
        'glpi_profilerights' => [
            'ON' => [
                'glpi_profilerights' => 'profiles_id',
                 'glpi_profiles' => 'id'
            ]
        ]
    ],
    'WHERE' => [
        'glpi_profilerights.name' => 'ticket',
        'glpi_profilerights.rights' => ['&', (READ | CREATE)];
    ]
]);
```

In this code example, the `READ | CREATE` make a bit sum, and the & operator compare the value at logical level

with the table.

Possible values for standard rights can be found in the `inc/define.php` file of GLPI:

```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);
```

Add a new right

Note

We *previously defined a property* `$rightname = 'computer'` on which we've automatically rights as `super-admin`. We will now create a specific right for the plugin.

First of all, let's create a new class dedicated to profiles management:

src/Profile.php

```php
<?php
namespace GlpiPlugin\Myplugin;

use CommonDBTM;
use CommonGLPI;
use Html;
use Profile as Glpi_Profile;

class Profile extends CommonDBTM
{
    public static $rightname = 'profile';

    static function getTypeName($nb = 0)
    {
        return __("My plugin", 'myplugin');
    }

    public function getTabNameForItem(CommonGLPI $item, $withtemplate = 0)
    {
        if (
            $item instanceof Glpi_Profile
            && $item->getField('id')
```

(continues on next page)

(continued from previous page)

```
        ) {
            return self::createTabEntry(self::getTypeName());
        }
        return '';
    }

    static function displayTabContentForItem(
        CommonGLPI $item,
        $tabnum = 1,
        $withtemplate = 0
    ) {
        if (
            $item instanceof Glpi_Profile
            && $item->getField('id')
        ) {
            return self::showForProfile($item->getID());
        }

        return true;
    }

    static function getAllRights($all = false)
    {
        $rights = [
            [
                'itemtype' => Superasset::class,
                'label'    => Superasset::getTypeName(),
                'field'    => 'myplugin::superasset'
            ]
        ];

        return $rights;
    }


    static function showForProfile($profiles_id = 0)
    {
        $profile = new Glpi_Profile();
        $profile->getFromDB($profiles_id);

        TemplateRenderer::getInstance()->display('@myplugin/profile.html.twig', [
            'can_edit' => self::canUpdate(),
            'profile'  => $profile,
            'rights'   => self::getAllRights()
        ]);
    }
}
```

Once again, display will be done from a Twig template:

templates/profile.html.twig

```twig
{% import "components/form/fields_macros.html.twig" as fields %}
<div class='firstbloc'>
    <form name="form" action="{{ "Profile"|itemtype_form_path }}" method="POST">
        <input type="hidden" name="id" value="{{ profile.fields['id'] }}">
        <input type="hidden" name="_glpi_csrf_token" value="{{ csrf_token() }}"> {# useless for GLPI 12 #}

        {% if can_edit %}
            <button type="submit" class="btn btn-primary mx-1" name="update" value="1">
                <i class="ti ti-device-floppy"></i>
                <span>{{ _x('button', 'Save') }}</span>
            </button>
        {% endif %}
    </form>
</div>
```

We declare a new tab on `Profile` object from our `init` function:

setup.php

```php
<?php

use Plugin;
use Profile;
use GlpiPlugin\Myplugin\Profile as MyPlugin_Profile;

function plugin_init_myplugin()
{
   ...

    Plugin::registerClass(MyPlugin_Profile::class, [
        'addtabon' => Profile::class
    ]);
}
```

And we tell installer to setup a minimal right for current profile (`super-admin`):

hook.php

```php
<?php

use GlpiPlugin\Myplugin\Profile as MyPlugin_Profile;
use ProfileRight;

function plugin_myplugin_install() {
   ...

   // add rights to current profile
   foreach (MyPlugin_Profile::getAllRights() as $right) {
      ProfileRight::addProfileRights([$right['field']]);
   }

   return true;
}
```

(continues on next page)

(continued from previous page)

```

function plugin_myplugin_uninstall() {
   ...

   // delete rights for current profile
   foreach (MyPlugin_Profile::getAllRights() as $right) {
      ProfileRight::deleteProfileRights([$right['field']]);
   }

}
```

Then, we can define rights from `Administration > Profiles` menu and can change the `$rightname` property of our class to `myplugin::superasset`.

Extending standard rights

If we need specific rights for our plugin, for example the right to perform associations, we must override the `getRights` function in the class defining the rights.

In defined above example of the `PluginMypluginProfile` class, we added a `getAllRights` method which indicates that the right `myplugin::superasset` is defined in the `PluginMypluginSuperasset` class. This one inherits from `CommonDBTM` and has a `getRights` method that we can override:

src/Superasset.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use CommonDBTM;
...

class Superasset extends CommonDBTM
{
    const RIGHT_ONE = 128;

    ...

    function getRights($interface = 'central')
    {
        // if we need to keep standard rights
        $rights = parent::getRights();

        // define an additional right
        $rights[self::RIGHT_ONE] = __("My specific rights", "myplugin");

        return $rights;
    }
}
```

5.12.14 Massive actions

GLPI massive actions allow applying modifications to a selection.

By default, GLPI proposes following actions:

- *Edit*: to edit fields that are defined in search options (excepted those where `massiveaction` is set to `false`)
- *Put in trashbin/Delete*

It is possible to declare *extra massive actions*.

To achieve that in your plugin, you must declare a hook in the `init` function:

setup.php

```php
<?php

function plugin_init_myplugin()
{
    ...

    $PLUGIN_HOOKS[Hooks::USE_MASSIVE_ACTION]['myplugin'] = true;
}
```

Then, in the `Superasset` class, you must add 3 methods:

- `getSpecificMassiveActions`: massive actions declaration.
- `showMassiveActionsSubForm`: sub-form display.
- `processMassiveActionsForOneItemtype`: handle form submit.

Here is a minimal implementation example:

src/Superasset.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use CommonDBTM;
use Html;
```

(continues on next page)

(continued from previous page)

```php
use MassiveAction;

class Superasset extends CommonDBTM
{
    ...

    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;
    }

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

                break;
        }

        return parent::showMassiveActionsSubForm($ma);
    }

    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);
```

(continues on next page)

(continued from previous page)

```
                        $ma->addMessage(__("Something went wrong"));
                    }
                }
                return;
            }

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

Note

Exercise: With the help of the official documentation on *massive actions*, complete in your plugin the above methods to allow the linking with a computer from "Super assets" massive actions.

You can display a list of computers with:

```
Computer::dropdown();
```

It is also possible to add massive actions to GLPI native objects. To achieve that, you must declare a `_MassiveActions` function in the `hook.php` file:

hook.php

```
<?php

use Computer;
use MassiveAction;
use GlpiPlugin\Myplugin\Superasset;

...

function plugin_myplugin_MassiveActions($type)
{
   $actions = [];
   switch ($type) {
      case Computer::class:
         $class = Superasset::class;
         $key   = 'DoIt';
         $label = __("plugin_example_DoIt", 'example');
         $actions[$class.MassiveAction::CLASS_ACTION_SEPARATOR.$key]
            = $label;

         break;
   }
   return $actions;
}
```

Sub form display and processing are done the same way as you did for your plugin itemtypes.

Note

Exercise: As the previous exercise, add a massive action to link a computer to a "Super asset" from the computer list.

Do not forget to use unique keys and labels.

5.12.15 Notifications

Warning

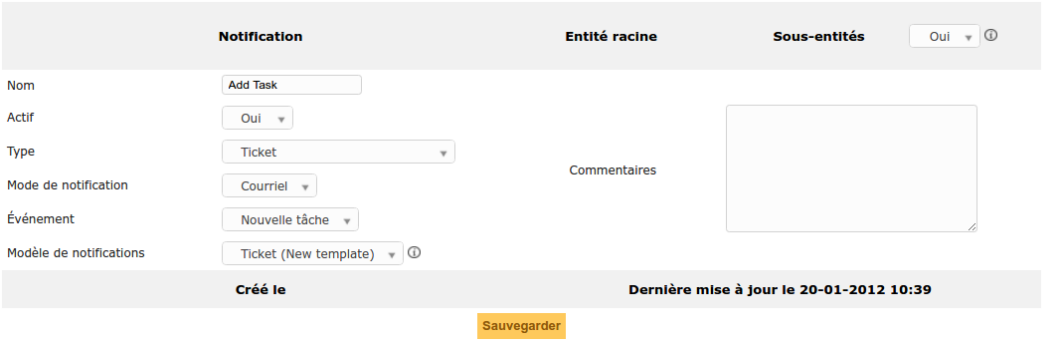
Access to an SMTP server is recommended; it must be properly configured in `Setup > Notifications` menu. On a development environment, you can install mailhog or mailcatcher which expose a local SMTP server and allow you to get emails sent by GLPI in a graphical interface.

Please also note that GLPI queues all notifications rather than sending them directly. The only exception to this is the test email notification. All "pending" notifications are visible in the `Administration > Notification queue` menu. You can send notifications immediately from this menu or by forcing the `queuednotification` automatic action.

The GLPI notification system allows sending alerts to the actors of a recorded event. By default, notifications can be sent by email or as browser notifications, but other channels may be available from plugins (or you can add your own one).

That system is divided in several classes:

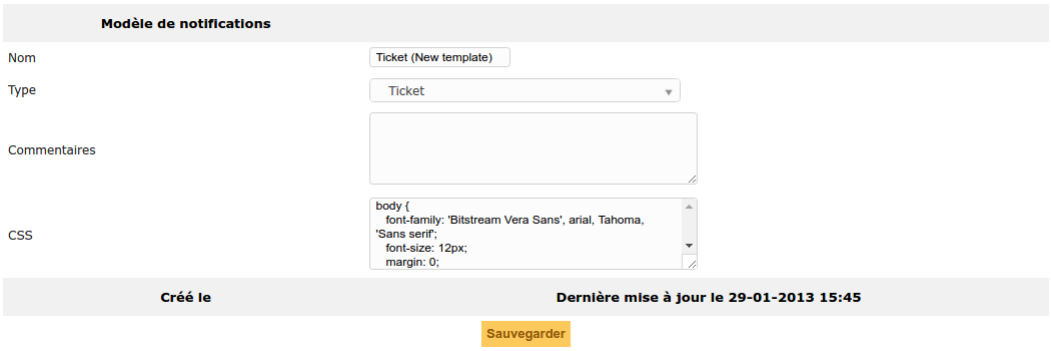
- `Notification`: the triggering item. It receives common data like name, if it is active, sending mode, event, content (`NotificationTemplate`), etc.



- **`NotificationTarget`: defines notification recipients.**

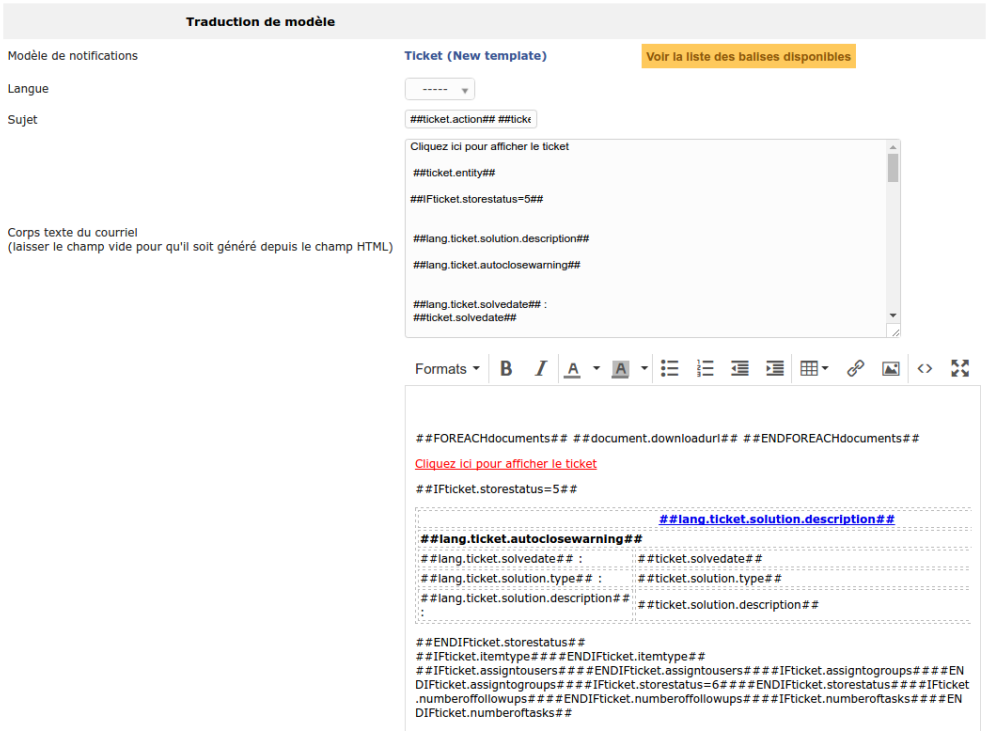
  It is possible to define recipients based on the triggering item (author, assignee) or static recipients (a specific user, all users of a specific group, etc).

- `NotificationTemplate`: notification templates are used to build the content, which can be chosen from Notification form. CSS can be defined in the templates and it receives one or more `NotificationTemplateTranslation` instances.



- `NotificationTemplateTranslation`: defines the translated template content. If no language is specified, it will be the default translation. If no template translation exists for a user's language, the default translation will be used.

  The content is dynamically generated with tags provided to the user and completed by HTML.



All of these notification-related object are natively managed by GLPI core and does not require any development intervention from us.

We can however trigger a notification execution via the following code:

```php
<?php

use NotificationEvent;

NotificationEvent::raiseEvent($event, $item);
```

The `event` key corresponds to the triggering event name defined in the `Notification` object and the `item` key to the triggering item. Therefore, the `raiseEvent` method will search the `glpi_notifications` table for an active line

with these 2 characteristics.

To use this trigger in our plugin, we will add a new class `PluginMypluginNotificationTargetSuperasset`. This ones targets our `Superasset` object. It is the standard way to develop notifications in GLPI. We have an itemtype with its own life and a notification object related to it.

src/NotificationTargetSuperasset.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use NotificationTarget;

class NotificationTargetSuperasset extends NotificationTarget
{

    function getEvents()
    {
        return [
            'my_event_key' => __('My event label', 'myplugin')
        ];
    }

    function getDatasForTemplate($event, $options = [])
    {
        $this->datas['##myplugin.name##'] = __('Name');
    }
}
```

We have to declare our `Superasset` object can send notifications in our `init` function:

setup.php

```php
<?php

use Plugin;
use GlpiPlugin\Myplugin\Superasset;

function plugin_init_myplugin()
{
   ...

    Plugin::registerClass(Superasset::class, [
        'notificationtemplates_types' => true
    ]);
}
```

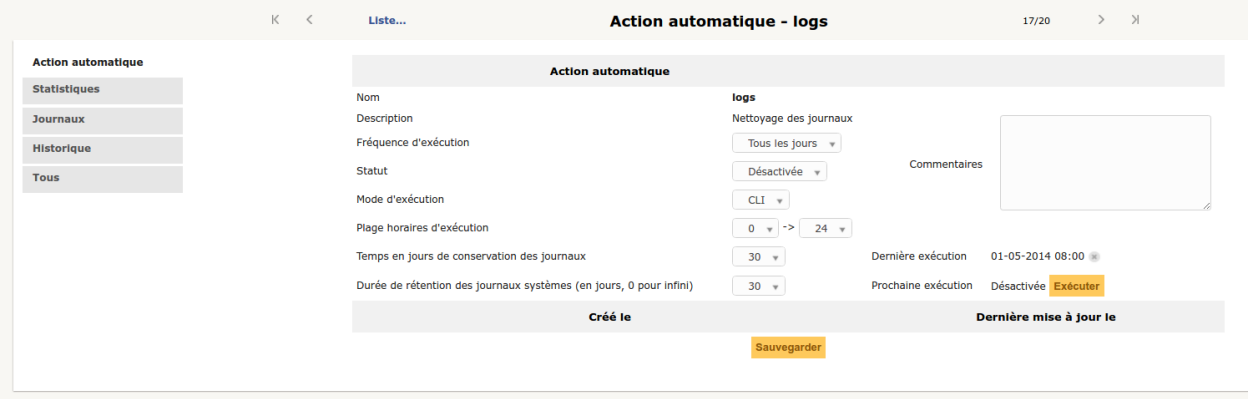
With this minimal code it's possible to create using the GLPI UI, a new notification targeting our `Superasset` itemtype and with the 'My event label' event and then use the `raiseEvent` method with these parameters.

> **Note**
>
> **Exercise**: Along with an effective sending test, you will manage installation and uninstallation of notification and related objects (templates, translations).

You can see an example (still incomplete) on *notifications in plugins documentation*.

5.12.16 Automatic actions

This GLPI feature provides a task scheduler executed silently from user usage (GLPI mode) or by the server in command line (CLI mode) via a call to the `front/cron.php` file of GLPI.



To add one or more automatic actions to our class, we will add those methods:

- `cronInfo`: possible actions for the class, and associated labels
- `cron*Action*`: a method for each action defined in `cronInfo`. Those are called to manage each action.

src/Superasset.php

```php
<?php

namespace GlpiPlugin\Myplugin;

use CommonDBTM;

class Superasset extends CommonDBTM
{
    ...

    static function cronInfo($name)
    {

        switch ($name) {
            case 'myaction':
                return ['description' => __('action desc', 'myplugin')];
        }
        return [];
    }

    static function cronMyaction($task = NULL)
    {
        // do the action

        return true;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

To tell GLPI that the automatic action exists, you just have to register it:

hook.php

```php
<?php

use CronTask;

function plugin_myplugin_install()
{

    ...

    CronTask::register(
        PluginMypluginSuperasset::class,
        'myaction',
        HOUR_TIMESTAMP,
        [
            'comment'   => '',
            'mode'      => \CronTask::MODE_EXTERNAL
        ]
    );
}
```

No need to manage uninstallation (*unregister*) as GLPI will handle that itself when the plugin is uninstalled.

5.12.17 Publishing your plugin

Catalog

When you consider your plugin is ready and covers a real need, you can submit it to the community.

The plugins catalog allows GLPI users to discover, download and follow plugins provided by the community as well as first-party plugins provided by Teclib'.

Just publish your code to an publicly accessible GIT repository (github, gitlab, ...) with an open source license of your choice and prepare an XML description file of your plugin. The XML file must follow this structure:

```xml
<root>
   <name>Displayed name</name>
   <key>System name</key>
   <state>stable</state>
   <logo>http://link/to/logo/with/dimensions/40px/40px</logo>
   <description>
      <short>
         <en>short description of the plugin, displayed on list, text only</en>
         <lang>...</lang>
      </short>
      <long>
         <en>short description of the plugin, displayed on detail, Markdown accepted</en>
         <lang>...</lang>
```

(continues on next page)

(continued from previous page)

```
      </long>
   </description>
   <homepage>http://link/to/your/page</homepage>
   <download>http://link/to/your/files</download>
   <issues>http://link/to/your/issues</issues>
   <readme>http://link/to/your/readme</readme>
   <authors>
      <author>Your name</author>
   </authors>
   <versions>
      <version>
         <num>1.0</num>
         <compatibility>10.0</compatibility>
         <download_url>http://link/to/your/download/glpi-myplugin-1.0.tar.bz2</download_
↪url>
      </version>
   </versions>
   <langs>
      <lang>en_GB</lang>
      <lang>...</lang>
   </langs>
   <license>GPL v2+</license>
   <tags>
      <en>
         <tag>tag1</tag>
      </en>
      <lang>
         <tag>tag1</tag>
      </lang>
   </tags>
   <screenshots>
      <screenshot>http://link/to/your/screenshot</screenshot>
      <screenshot>http://link/to/your/screenshot</screenshot>
      <screenshot>...</screenshot>
   </screenshots>
</root>
```

To market this plugin to a wide range of users, you should add a detailed description in several languages and provide screenshots that represent your plugin.

Finally, submit your XML file on the dedicated page of the plugins catalog (registration is required).

Note

Path to plugin XML file must display the raw XML file itself. For example, the following URL for the *exmple* plugin would be incorrect:

```
https://github.com/pluginsGLPI/example/blob/main/example.xml
```

The correct one (use Github UI *raw* button) would be:

```
https://raw.githubusercontent.com/pluginsGLPI/example/refs/heads/main/example.xml
```

Teclib' will receive a notification for this submission and after some checks, will activate the publication on the catalog.

Marketplace

By following these steps and recommendations, you will be able to make your plugin available on the GLPI Marketplace, thus offering users simplified installation and updates. We would like to thank you for this contribution, which helps enrich the GLPI ecosystem for the entire community.

1. **Preparation**:

a. Your plugin archive should contain a directory with a name corresponding to the plugin's technical name. All your plugin's files should be placed in this directory.

   Example:

   for a plugin whose `plugin_init_` function is `plugin_init_oauthimap` in `setup.php`, the technical name of its directory must be `oauthimap`. The plugin's files should be located inside a directory named `oauthimap`.

b. Make sure your XML file contains a <key> element that exactly matches this directory name (no spaces, no accents, no uppercase letters).

   Example: <key>oauthimap</key>

c. In the <versions> section of your XML file, for each version of your plugin (with version number and compatibility), add a `download_url` tag containing the URL where the plugin archive can be downloaded.

Example:

```
<versions>
  <version>
    <num>1.0</num>
    <compatibility>~10.0.0</compatibility>
    <download_url>https://link/to/your/plugin/file-1.0.tar.gz</download_url>
  </version>
</versions>
```

d. In the <versions> section of your XML file, for each version of your plugin the `compatibility` tag value must correspond to a GLPI version constraint in a format compatible with the composer API.

   Example: <compatibility>~10.0.7</compatibility>

1. **Public Access**:

- Make sure the URL of the XML file and the plugin archive download URL are publicly accessible.
- Ensure that the plugin archive is properly structured and downloadable using the URL provided in the XML file.

Technical Requirements and Recommendations

1. **Compliance with Coding Standards**:

- Follow the recommendations in the GLPI Developer Documentation: GLPI Developer Documentation
- Ensure your code complies with GLPI coding standards and does not trigger errors from tools like phpcs.

2. **Code Security and Quality**:

- Avoid raw SQL queries. Always use GLPI framework methods (see *Querying* and *Updating*) — **this is mandatory starting from GLPI 11**.
- Use Twig for templating.
- Properly enforce permissions in all front-end (front/*) and AJAX (ajax/*) files — **this is mandatory**.
- The plugin may be rejected if it contains backdoors or obvious security flaws.

3. **Compatibility and Updates**:

- Make sure your plugin is compatible with a maintained version of GLPI.
- Keep your plugin up to date to ensure continued compatibility with future GLPI versions.

Submission Process

> **Note**
>
> Before continuing, your plugin must be published on the plugins catalog, see above.

1. **Validation and Approval**:

- By default, plugins accepted on the Plugins Website are not automatically available on the Marketplace. For security and relevance reasons, the GLPI team must review key technical aspects before approving Marketplace availability.
- If your plugin is already listed on the Plugins Website and you want to distribute it on the Marketplace, please send an email to the GLPI team at glpi@teclib.com.
- Depending on the results of the review, the team may approve the plugin for availability on the on-premise GLPI Marketplace, and/or the Cloud instance Marketplace (which has stricter security requirements).

2. **Lifecycle and Maintenance**

    - Ongoing Monitoring:
    - After approval and publication, regularly monitor your plugin's performance and security to ensure continued compliance with GLPI requirements.
    - Plugin Deactivation:
    - The GLPI team reserves the right to deactivate the plugin from the Marketplace if, at any point, it no longer meets requirements, causes a major bug, or presents a critical security vulnerability.

Therefore, it is crucial to maintain your plugin and promptly address any reported issues.

5.12.18 Miscellaneous

Querying database

Rely on *DBmysqlIterator*. It provides an exhaustive `query builder`.

```php
<?php


// => SELECT * FROM `glpi_computers`
$iterator = $DB->request(['FROM' => 'glpi_computers']);
foreach ($ierator as $row) {
    //... work on each row ...
}

$DB->request([
    'FROM' => ['glpi_computers', 'glpi_computerdisks'],
    'LEFT JOIN' => [
        'glpi_computerdisks' => [
            'ON' => [
                'glpi_computers' => 'id',
                'glpi_computerdisks' => 'computer_id'
            ]
        ]
    ]
]);
```

Dashboards

Since GLPI 9.5, dashboards are available from:

- Central page
- Assets menu
- Assistance menu
- Ticket search results (mini dashboard)

This feature is split into several concepts - sub classes:

- a placement grid (`Glpi\Dashboard\Grid`)
- a widget collection (`Glpi\Dashboard\Widget`) to graphically display data
- a data provider collection (`Glpi\Dashboard\Provider`) that queries the database
- rights (`Glpi\Dashboard\Right`) on each dashboard
- filters (`Glpi\Dashboard\Filter`) that can be displayed in a dashboard header and impacting providers.

With these classes, we can build a dashboard that will display cards on its grid. A card is a combination of a widget, a data provider, a place on grid and various options (like a background colour for example).

5.12.19 Completing existing concepts

From your plugin, you can complete these concepts with your own data and code.

setup.php

```php
<?php

use Glpi\Plugin\Hooks;
use GlpiPlugin\Myplugin\Dashboard;

function plugin_init_myplugin()
{
    ...

    // add new widgets to the dashboard
    $PLUGIN_HOOKS[Hooks::DASHBOARD_TYPES]['myplugin'] = [
        Dashboard::class => 'getTypes',
    ];

    // add new cards to the dashboard
    $PLUGIN_HOOKS[Hooks::DASHBOARD_CARDS]['myplugin'] = [
        Dashboard::class => 'getCards',
    ];
}
```

We will create a dedicated class for our dashboards:

src/Dashboard.php

```php
<?php

namespace GlpiPlugin\Myplugin;

class Dashboard
{
    static function getTypes()
    {
        return [
            'example' => [
                'label'    => __("Plugin Example", 'myplugin'),
                'function' => __class__ . "::cardWidget",
                'image'    => "https://via.placeholder.com/100x86?text=example",
            ],
            'example_static' => [
                'label'    => __("Plugin Example (static)", 'myplugin'),
                'function' => __class__ . "::cardWidgetWithoutProvider",
                'image'    => "https://via.placeholder.com/100x86?text=example+static",
            ],
        ];
    }

    static function getCards($cards = [])
    {
        if (is_null($cards)) {
            $cards = [];
        }
        $new_cards =  [
            'plugin_example_card' => [
```

(continues on next page)

(continued from previous page)

```
                'widgettype'   => ["example"],
                'label'        => __("Plugin Example card"),
                'provider'     => "PluginExampleExample::cardDataProvider",
            ],
            'plugin_example_card_without_provider' => [
                'widgettype'   => ["example_static"],
                'label'        => __("Plugin Example card without provider"),
            ],
            'plugin_example_card_with_core_widget' => [
                'widgettype'   => ["bigNumber"],
                'label'        => __("Plugin Example card with core provider"),
                'provider'     => "PluginExampleExample::cardBigNumberProvider",
            ],
        ];

        return array_merge($cards, $new_cards);
    }

    static function cardWidget(array $params = [])
    {
        $default = [
            'data'  => [],
            'title' => '',
            // this property is "pretty" mandatory,
            // as it contains the colors selected when adding widget on the grid send
            // without it, your card will be transparent
            'color' => '',
        ];
        $p = array_merge($default, $params);

        // you need to encapsulate your html in div.card to benefit core style
        $html = "<div class='card' style='background-color: {$p["color"]};'>";
        $html.= "<h2>{$p['title']}</h2>";
        $html.= "<ul>";
        foreach ($p['data'] as $line) {
            $html.= "<li>$line</li>";
        }
        $html.= "</ul>";
        $html.= "</div>";

        return $html;
    }

    static function cardWidgetWithoutProvider(array $params = [])
    {
      $default = [
         // this property is "pretty" mandatory,
         // as it contains the colors selected when adding widget on the grid send
         // without it, your card will be transparent
         'color' => '',
      ];
      $p = array_merge($default, $params);
```

(continues on next page)

(continued from previous page)

```
82
83      // you need to encapsulate your html in div.card to benefit core style
84      $html = "<div class='card' style='background-color: {$p["color"]};'>
85                  static html (+optional javascript) as card is not matched with a data␣
    ↪provider
86                  <img src='https://www.linux.org/images/logo.png'>
87              </div>";
88
89      return $html;
90   }
91
92    static function cardBigNumberProvider(array $params = [])
93    {
94        $default_params = [
95            'label' => null,
96            'icon'  => null,
97        ];
98        $params = array_merge($default_params, $params);
99
100       return [
101           'number' => rand(),
102           'url'    => "https://www.linux.org/",
103           'label'  => "plugin example - some text",
104           'icon'   => "fab fa-linux", // font awesome icon
105       ];
106   }
107}
```

A few explanations on those methods:

- `getTypes()`: define available widgets for cards and methods to call for display.
- `getCards()`: define available cards for dashboards (when added to the grid). As previously explained, each is defined from a label, widget and optional data provider (from core or your plugin) combination
- `cardWidget()`: use provided parameters to display HTML. You are free to delegate display to a Twig template, and use your favourite JavaScript library.
- `cardWidgetWithoutProvider()`: almost the same as the `cardWidget()`, but does not use parameters and just returns a static HTML.
- `cardBigNumberProvider()`: provider and expected return example when grid will display card.

5.12.20 Display your own dashboard

GLPI dashboards system is modular and you can use it in your own displays.

```
1 <?php
2
3 use Glpi\Dashboard\Grid;
4
5 $dashboard = new Grid('myplugin_example_dashboard', 10, 10, 'myplugin');
6 $dashboard->show();
```

By adding a context (`myplugin`), you can filter dashboards available in the dropdown list at the top right of the grid. You will not see GLPI core ones (central, assistance, etc.).

Translating your plugins

In many places in current document, code exmaples takes care of using gettext GLPI notations to display strings to users. Even if your plugin will be private, it is a good practice to keep this gettext usage.

See *developper guide translation documentation* for more explanations and list of PHP functions that can be used.

- On your local instance, you can use software like poedit to manage your translations.
- You can also rely on online services like Transifex or Weblate (both are free for open source projects).

If you have used the Empty plugin skeleton, you will benefit from command line tools to manage your locales:

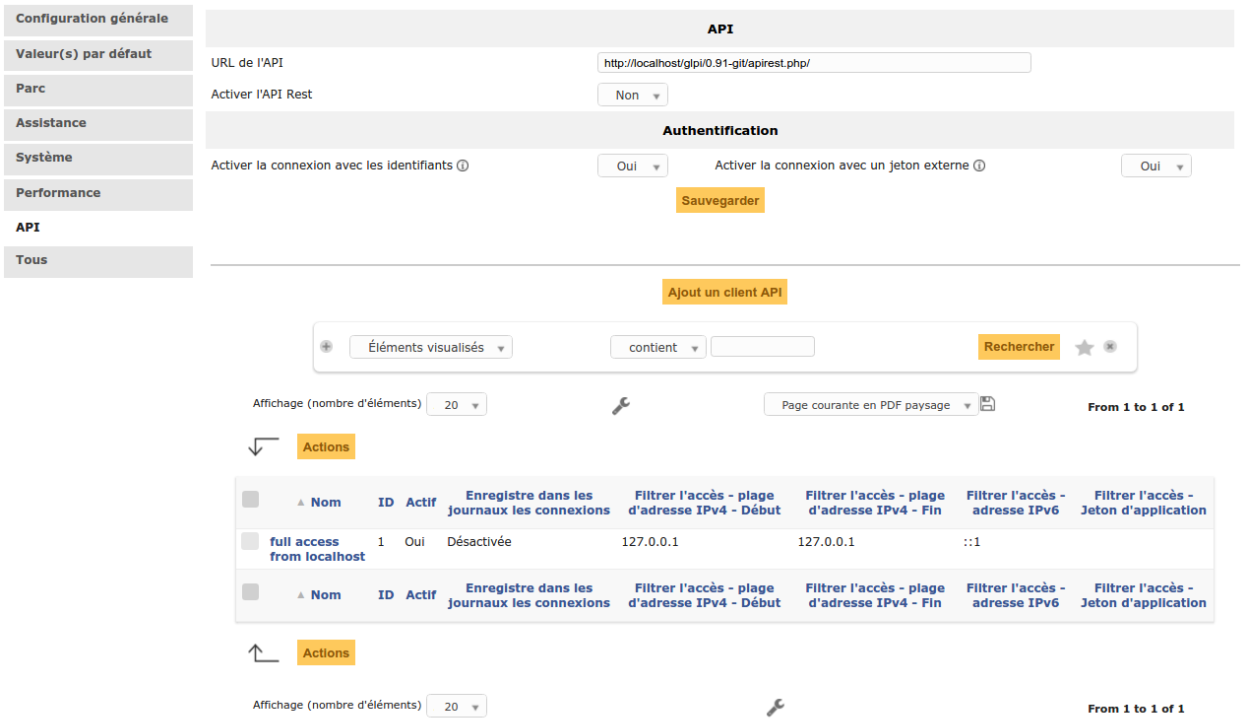
```
# extract strings to translate from your source code
# and put them in the locales/myplugin.pot file
vendor/bin/extract-locales
```

Warning

It is possible your translations are not updated after compiling MO files, a restart of your PHP (or web server, depending on your configuration) may be required.

5.12.21 REST API

Since GLPI (since 9.1 release) has an external API in REST format. An XMLRPC format is also still available, but is deprecated.



Configuration

For security reasons, API is disabled bu default. From the `Setup > General, API tab` menu, you can enable it.

It's available from your instance at:

- `http://path/to/glpi/apirest.php`
- `http://path/to/glpi/apixmlrpc.php`

The first link includes an integrated documentation when you access it from a simple browser (a link is provided as soon as the API is active).

For the rest of the configuration:

- login allows to use `login` / `password` as well as web interface
- token connection use the token displayed in user preferences

- API clients allow to limit API access from some IP addresses and log if necessary. A client allowing access from any IP is provided by default.

You can use the API usage bootstrap. This one is written in PHP and relies on Guzzle library to handle HTTP requests.

By default, it does a connection with login details defined in the `config.inc.php` file (that you must create by copying the `config.inc.example` file).

> **Warning**
>
> Make sure the script is working as expected before continuing.

API usage

To learn this part, with the help of integrated documentation (or latest stable GLPI API documentation on github), we will do several exercises:

> **Note**
>
> **Exercise**: Test a new connection using GLPI user external token

> **Note**
>
> **Exercise**: Close the session at the end of your script.

> **Note**
>
> **Exercise**: Simulate computer life cycle:
>
> - add a computer and some volumes (`Item_Disk`),

- edit several fields,
- add commercial and administrative information (`Infocom`),
- display its detail in a PHP page,
- put it in the trashbin,
- and then remove it completely.

Note

Exercise: Retrieve computers list and display them an HTML array. The *endpoint* to use us "Search items". If you want to display columns labels, you will have to use the "List searchOptions" *endpoint*.

5.13 Javascript

5.13.1 Vue.js

Please refer to *the core Vue developer documentation first*.

Plugins that wish to use custom Vue components must implement their own webpack config to build the components and add them to the *window.Vue.components* object.

Sample webpack config (derived from the config used in GLPI itself for Vue):

```
const webpack = require('webpack');
const path = require('path');
const VueLoaderPlugin = require('vue-loader').VueLoaderPlugin;

const config = {
    entry: {
        'vue': './js/src/vue/app.js',
    },
    externals: {
        // prevent duplicate import of Vue library (already done in ../../public/build/
↪vue/app.js)
        vue: 'window _vue',
    },
    output: {
        filename: 'app.js',
        chunkFilename: "[name].js",
        chunkFormat: 'module',
        path: path.resolve(__dirname, 'public/build/vue'),
        publicPath: '/public/build/vue/',
        asyncChunks: true,
        clean: true,
    },
    module: {
```

(continues on next page)

(continued from previous page)

```
        rules: [
            {
                // Vue SFC
                test: /\.vue$/,
                loader: 'vue-loader'
            },
            {
                // Build styles
                test: /\.css$/,
                use: ['style-loader', 'css-loader'],
            },
        ]
    },
    plugins: [
        new VueLoaderPlugin(), // Vue SFC support
        new webpack.ProvidePlugin(
            {
                process: 'process/browser'
            }
        ),
        new webpack.DefinePlugin({
            __VUE_OPTIONS_API__: false, // We will only use composition API
            __VUE_PROD_DEVTOOLS__: false,
        }),
    ],
    resolve: {
        fallback: {
            'process/browser': require.resolve('process/browser.js')
        },
    },
    mode: 'none', // Force 'none' mode, as optimizations will be done on release process
    devtool: 'source-map', // Add sourcemap to files
    stats: {
        // Limit verbosity to only usefull information
        all: false,
        errors: true,
        errorDetails: true,
        warnings: true,

        entrypoints: true,
        timings: true,
    },
    target: "es2020"
};

module.exports = config
```

Note the use of the `externals` option. This will prevent webpack from including Vue itself when building your components since it is already imported by the bundle in GLPI itself. The core GLPI bundle sets `window._vue` to the vue module and the plugin's externals option will map any imports from 'vue' to that. This will drastically reduce the size of your imports.

For your entrypoint, it is mostly the same as the core GLPI one except you should use the `defineAsyncComponent`

method in `window.Vue` instead of importing it from Vue itself.

Example entrypoint:

```
// Require all Vue SFCs in js/src directory
const component_context = import.meta.webpackContext('.', {
    regExp: /\.vue$/i,
    recursive: true,
    mode: 'lazy',
    chunkName: '/vue-sfc/[request]'
});
const components = {};
component_context.keys().forEach((f) => {
    const component_name = f.replace(/^\.\/(.+)\.vue$/, '$1');
    components[component_name] = {
        component: window.Vue.defineAsyncComponent(() => component_context(f)),
    };
});
// Save components in global scope
window.Vue.components = Object.assign(window.Vue.components || {}, components);
```

To keep your components from colliding with core components or other plugins, it you should organize them inside the *js/src/Plugin/Yourplugin* folder. This will ensure plugin components are registered as `Plugin/Yourplugin/YourComponent`. You can organize components further with additional subfolders.

(cc) BY-NC-ND

CHAPTER

SIX

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
```

(continues on next page)

(continued from previous page)

```
#<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
      Require all denied
   </IfModule>
   <IfModule !mod_authz_core.c>
      # Apache 2.2
      Order Deny,Allow
      Deny from All
   </IfModule>
```

(continues on next page)

(continued from previous page)

```
</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';
// "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"
```

(continues on next page)

(continued from previous page)

```
// "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');
```

CHAPTER

SEVEN

UPGRADE GUIDES

The upgrade guides are intended to help you adapt your plugins to the changes introduced in the new versions of GLPI.

> **Note**
>
> Only the most important changes and those requiring support are documented here. If you are having trouble migrating your code, feel free to suggest a documentation update.

7.1 Upgrade to GLPI 11.0

7.1.1 Removal of input variables auto-sanitize

Prior to GLPI 11.0, PHP superglobals `$_GET`, `$_POST` and `$_REQUEST` were automatically sanitized. It means that SQL special characters were escaped (prefixed by a \), and HTML special characters <, > and & were encoded into HTML entities. This caused issues because it was difficult, for some pieces of code, to know if the received variables were "secure" or not.

In GLPI 11.0, we removed this auto-sanitization, and any data, whether it comes from a form, the database, or the API, will always be in its raw state.

Protection against SQL injection

Protection against SQL injection is now automatically done when DB query is crafted.

All the `addslashes()` usages that were used for this purpose have to be removed from your code.

```
- $item->add(Toolbox::addslashes_deep($properties));
+ $item->add($properties);
```

Protection against XSS

HTML special characters are no longer encoded automatically. Even existing data will be seamlessly decoded when it will be fetched from database. Code must be updated to ensure that all dynamic variables are correctly escaped in HTML views.

Views built with `Twig` templates no longer require usage of the `|verbatim_value` filter to correctly display HTML special characters. Also, Twig automatically escapes special characters, which protects against XSS.

```
- <p>{{ content|verbatim_value }}</p>
+ <p>{{ content }}</p>
```

Code that outputs HTML code directly must be adapted to use the `htmlescape()` function.

```
- echo '<p>' . $content . '</p>';
+ echo '<p>' . htmlescape($content) . '</p>';
```

Also, code that ouputs javascript must be adapted to prevent XSS by escaping both the HTML code with the `htmlescape()` function and the JS variables with the `jsescape()` function.

```
echo '
    <script>
-       $(body).append("<p>' . $content . '</p>");
+       $(body).append("' . jsescape('<p>' . htmlescape($content) . '</p>') . '");
    </script>
';
```

Note

Both the `htmlescape()` and the `jsescape()` functions have been added to ease the migration to GLPI 11.0 but will be deprecated and removed when the GLPI HTML and JS code will be completely moved into `Twig` templates and JS files.

Query builder usage

Since it has been implemented, internal query builder (named `DBMysqlIterator`) do accept several syntaxes; that make things complex:

1. conditions (including table name as `FROM` array key) as first (and only) parameter.
2. table name as first parameter and condition as second parameter,
3. raw SQL queries,

The most used and easiest to maintain was the first. The second has been deprecated and the third has been prohibited or security reasons.

If you were using the second syntax, you will need to replace as follows:

```
- $iterator = $DB->request('mytable', ['field' => 'condition']);
+ $iterator = $DB->request(['FROM' => 'mytable', 'WHERE' => ['field' => 'condition']]);
```

Using raw SQL queries must be replaced with query builder call, among other to prevent syntax issues, and SQL injections; please refer to *Querying*.

7.1.2 Changes related to web requests handling

In GLPI 11.0, all the web requests are now handled by a unique entry point, the `/public/index.php` script. This allowed us to centralize a large number of things, including GLPI's initialization mechanics and error management.

Removal of the `/inc/includes.php` script

All the logic that was executed by the inclusion of the `/inc/includes.php` script is now made automatically. Therefore, it is no longer necessary to include it, even if it is still present to ease the migration to GLPI 11.0.

```
- include("../../../inc/includes.php");
```

Resource access restrictions

In GLPI 11.0, we restrict the resources that can be accessed through a web request.

To ease the migration to GLPI 11.0, we still support public access to the PHP scripts located in the `/ajax`, `/front` and `/report` directories, and their URL remains unchanged.

All static assets or other PHP scripts that must be accessible through a web request must be moved in the `/public` directory. The `/public` part of the path must not be present in their URL, for instance:

- the URL of the `/public/css/styles.css` stylesheet of your plugin will be `/plugins/myplugin/css/styles.css`;
- the URL of the `/public/mypluginapi.php` script of your plugin will be `/plugins/myplugin/mypluginapi.php`.

Legacy scripts access policy

By default, the access to any PHP script will be allowed only to authenticated users. If you need to change this default policy for some of your PHP scripts, you will need to do this in your plugin `init` function, using the `Glpi\Http\Firewall::addPluginStrategyForLegacyScripts()` method.

```php
<?php

use Glpi\Http\Firewall;

function plugin_init_myplugin() {
    Firewall::addPluginStrategyForLegacyScripts('myplugin', '#^/front/faq.php$#',␣
↪Firewall::STRATEGY_FAQ_ACCESS);
    Firewall::addPluginStrategyForLegacyScripts('myplugin', '#^/front/dashboard.php$#',␣
↪Firewall::STRATEGY_CENTRAL_ACCESS);
}
```

The following strategies are available:

- `Firewall::STRATEGY_NO_CHECK`: no check is done, anyone can access your script, even unauthenticated users;
- `Firewall::STRATEGY_AUTHENTICATED`: only authenticated users can access your script, it is the default strategy for all PHP scripts;
- `Firewall::STRATEGY_CENTRAL_ACCESS`: only users with access to the standard interface can access your script;
- `Firewall::STRATEGY_HELPDESK_ACCESS`: only users with access to the simplified interface can access your script;
- `Firewall::STRATEGY_FAQ_ACCESS`: only users with a read access to the FAQ will be allowed to access your script, unless the FAQ is configured to be public.

Stateless endpoints

By default, GLPI will automatically start the PHP session, and use a session cookie to share the current session ID between web requests. If there is no active session, it will redirect the client to the login page. This behaviour should be disabled for stateless endpoints, such as APIs endpoints. To do this, you will need to call the `\Glpi\Http\SessionManager::registerPluginStatelessPath()` method from the `boot` hook of your plugin, located in the `setup.php` file.

```php
<?php
```

(continues on next page)

(continued from previous page)

```
use Glpi\Http\SessionManager;

function plugin_init_myplugin() {
    SessionManager::registerPluginStatelessPath('myplugin', '#^/front/api.php/#');
}
```

Handling of response codes and early script exit

Usage of the `exit()`/`die()` language construct is now discouraged as it prevents the execution of routines that might take place after the request has been executed. Also, due to a PHP bug (see https://bugs.php.net/bug.php?id=81451), the usage of the `http_response_code()` function will produce unexpected results, depending on the server environment.

In the case they were used to exit the script early due to an error, you can replace them by throwing an exception. Any exception thrown will now be caught correctly and forwarded to the error handler. If this exception is thrown during the execution of a web request, the GLPI error page will be shown, unless this exception is handled by a specific routine.

```
if ($item->getFromDB($_GET['id']) === false) {
-    http_response_code(404);
-    exit();
+    throw new \Glpi\Exception\Http\NotFoundHttpException();
}
```

In case the `exit()`/`die()` language construct was used to just ignore the following line of code in the script, you can replace it with a `return` instruction.

```
if ($action === 'foo') {
    // specific action
    echo "foo action executed";
-    exit();
+    return;
}

MypluginItem::displayFullPageForItem($_GET['id']);
```

Crafting plugins URLs

We changed the way to handle URLs to plugin resources so that they no longer need to reflect the location of the plugin on the file system. For instance, the same URL could be used to access a plugin file whether it was installed manually in the `/plugins` directory or via the marketplace.

To maintain backwards compatibility with previous behavior, we will continue to support URLs using the `/marketplace` path prefix. However, their use is deprecated and may be removed in a future version of GLPI.

The `Plugin::getWebDir()` PHP method has been deprecated.

```
- $path = Plugin::getWebDir('myplugin', false) . '/front/myscript.php';
+ $path = '/plugins/myplugin/front/myscript.php';

- $path = Plugin::getWebDir('myplugin', true) . '/front/myscript.php';
+ $path = $CFG_GLPI['root_doc'] . '/plugins/myplugin/front/myscript.php';
```

The `GLPI_PLUGINS_PATH` javascript variable has been deprecated.

```
- var url = CFG_GLPI.root_doc + '/' + GLPI_PLUGINS_PATH.myplugin + '/ajax/script.php';
+ var url = CFG_GLPI.root_doc + '/plugins/myplugin/ajax/script.php';
```

The `get_plugin_web_dir` Twig function has been deprecated.

```
- <form action="{{ get_plugin_web_dir('myplugin') }}/front/config.form.php" method="post
↪">
+ <form action="{{ path('/plugins/myplugin/front/config.form.php') }}" method="post">
```

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/latest/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/latest/source/plugins/hooks.rst, line 658.)

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/latest/source/plugins/hooks.rst, line 664.)

Todo

Write documentation for this hook.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/glpi-developer-documentation/checkouts/latest/source/plugins/hooks.rst, line 686.)