# Common

Some useful classes and functionalities are provided in a common part, which any component may use. Typically this is reusable functions, or transverse functions.

## Localization

In order to make the application multi-language, any data displayed on a page which does not come from the Database must be localized.

Each component can define localized strings, located in the sub-directory *locale* of the component.

Localized strings are provided as map between keys (typically the string in English) and localized value (the translation).

Localized strings are case insensitive, but keys and translations can provide indications where the different words are, using the ~ character.

For example, with given map:

~add ~user=~ajouter un ~utilisateur

If a screen request for "Add User", it will be translated into "Ajouter un Utilisateur": meaning the capital letters are put according to the indicated words, and the capital letters given in the requested string (so there is no need to define 2 different mapping, one with capital letters, one without).

The language used is kept in the session of the user, but also in a cookie in order to keep the language of the user over sessions. If no information is available, it will detect the preferred language set in the browser of the user. If still no information is available, English will be used by default.

If translations are not provided in the language of the user, by default English will be used. Meaning English translations are mandatory to be provided, while other languages are optional (and can be done later on, when we have a translator).

Localized strings are split into *namespaces*, to avoid conflict (the same word may have different translations depending on the context). By default localized strings of a component are stored under the namespace having the same name as the component (i.e. user\_management component will have the namespace user\_management). To ease the usage, when we are in a page of a component, the default namespace is the namespace of this component, so no need to specify the namespace. However it is still possible to specify it, and so to access strings defined by other components.

A *common* namespace is also provided, containing most common words which may be used by any component (for example words like “cancel”, “add”, “remove”…)

D:\_PN\student_soft\repositories\lecousin\doc\system\diagrams\Common_Locale.wmf

The Locale class contains two static attributes: language containing the language code of the user, and current\_component containing the default namespace to be used, in case no namespace if provided to the functions.

It provides the following functions:

|  |  |
| --- | --- |
| **Method** | **Description** |
| get\_string(component, key) | Return the translation for the given key in the given component |
| load(component) | Load the translations provided by the given component, in the language of the user.  This is equivalent to the call load\_file(component, “component/”+component+”/locale/”); |
| load\_file(namespace, path) | Load in the given namespace, the translations located in the given path, for the language of the user.  This is equivalent to the call load\_path(namespace, path+language, “UTF-8”); |
| load\_path(namespace, path, charset) | Load the translations located in the file path in the given namespace using the given charset (by default charset UTF-8 is used) |

To make easier the request of a translation in PHP code, 2 functions out of the Locale class are provided:

function get\_locale()

This function will call the function Locale::get\_string. If called with 1 argument, the argument is the key, and the current\_component is used as namespace. If called with 2 arguments, they will be used as namespace and key.

function locale()

Do exactly the same as get\_locale but print the translation to the output using echo.

The translations are mainly used when generating content in PHP, but in few cases, it may be useful also to be loaded dynamically from JavaScript: the JavaScript can remain as a static resource, and load dynamically the translation for the few words it needs.

Translations are available using URL /locale/<component\_name>/

By default, the translations in *common* are automatically loaded.

The JavaScript file common/js/locale.js implements similar functions as the PHP class Locale, but to be used in JavaScript.

## DataBase

As the software will be mainly a Database, some functionalities are provided to access the Database, to know the data model, and to create pages to display and edit data. Also the Database should not be accessed directly but by using the functionalities provided, which will ensure security checks.

To access directly the Database, we will use a very basic abstraction layer, providing basic functionalities as executing a SQL query, and get the result of it, or the error if something wrong occurred. This abstraction layer is the class *DataBaseSystem*, accessed through static class *Database*. One implementation is provided for MySQL: *DataBaseSystem\_MySQL*.

However, in most of the cases the Database must not be accessed directly, for security purpose, but through one of the 2 methods:

* The class *SQLQuery*: this class provides functions to build a SQL request (select fields on a table, join tables, update fields, insert, remove…)
* The class *DataModel* provided by the “Data Model” component, which will be discuss later.

Those classes are implementing security checks, to ensure what will be done on the database is allowed for the user. This will allow the developers not to really take care about permissions in their code, as the permissions will be automatically checked by those functionalities, and thus ensure a developer does not forget to make all the needed security checks.

This will be discussed in more details in the chapter of the “Data Model” component.

## Concurrent access

Due to the nature of the application (web), several users can access to data concurrently (at the same time). In order to keep consistency, and avoid mistakes, we need a system to ensure two users will not modify the same data at the same time.

For this, when a user access a page where it can edit data, the page will first need to lock the data which can be edited. Then, to keep this lock, the page will need to regularly inform that the lock is still active. Indeed, to ensure data will not be locked indefinitely, a lock will expire after 10 minutes, if it was not extended.

To keep a lock active, while activity of user is detected (mouse move, click..) we will regularly ask to extend the expiration time of the lock. If after 5 minutes of inactivity, we will display a popup, asking the user to confirm he is still active. If the user does not answer within a minute, the lock will be automatically released, and the user redirected to another page.

Also, when the user save data, the page or service must check the user has still a lock active on the data to save.

This mechanism is provided by the component *application*: the JavaScript databaselock.js handles the mechanism to detect user activity or inactivity.

On PHP side, the file common/DataBaseLock.inc provides functionalities to get a lock, release a lock, check the validity of a lock, check if the current user has a lock on given data…