**Students Database**

Technical Design

**Document Revision History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ver. | Date | Author | Reviewed by | Approved by | Changes |
| 0.1 | 01/01/2011 | Someone | Someone else | Someone else | Initial draft |
|  |  |  |  |  |  |

Table of Contents

[1 General Design 3](#_Toc353200161)

[1.1 Objectives 3](#_Toc353200162)

[1.2 Structure 3](#_Toc353200163)

[2 Common 4](#_Toc353200164)

[2.1 Localization 4](#_Toc353200165)

[2.2 DataBase 5](#_Toc353200166)

[3 General Components 5](#_Toc353200167)

[3.1 Authentication 5](#_Toc353200168)

[3.1.1 Responsibilities 5](#_Toc353200169)

[3.1.2 Functionalities 6](#_Toc353200170)

[3.2 User Management 6](#_Toc353200171)

[3.2.1 Responsibilities 6](#_Toc353200172)

[3.2.2 Functionalities 6](#_Toc353200173)

[3.2.3 Data 7](#_Toc353200174)

[3.3 People 8](#_Toc353200175)

[3.3.1 Responsibilities 8](#_Toc353200176)

[3.3.2 Functionalities 8](#_Toc353200177)

[3.3.3 Data 8](#_Toc353200178)

[3.4 Storage 8](#_Toc353200179)

[3.4.1 Responsibilities 8](#_Toc353200180)

[3.4.2 Functionalities 8](#_Toc353200181)

[3.4.3 Data 8](#_Toc353200182)

[3.5 Contact 9](#_Toc353200183)

[3.5.1 Responsibilities 9](#_Toc353200184)

[3.5.2 Functionalities 9](#_Toc353200185)

[3.5.3 Data 9](#_Toc353200186)

[4 Selection Components 9](#_Toc353200187)

[5 Education Components 9](#_Toc353200188)

[6 Training Components 9](#_Toc353200189)

[7 External Relations Components 9](#_Toc353200190)

[8 Finance Components 9](#_Toc353200191)

# General Design

## Objectives

The two main objectives are:

1. Good separation of responsibilities into small components, to ease the development by several people, and making easier to build the application step by step.
2. Keep the design simple, so anyone can easily be part of the development, without long training

To reach the first objective, each component will:

* be located into a separate directory
* have the responsibility of his own tables and data in the database, and provide functionalities to access data for other components
* defines its own web pages, if applicable
* declare its dependencies, and cyclic dependency will automatically generate an error
* not be allowed to directly access tables which is not under its responsibility

To reach the second objective:

* we will not use a complete framework which may be long to learn, but may use small part of it or small libraries at some points to avoid doing something that already exists (like importing Excel files…)
* but still we will implement few principles usually provided by frameworks, as good design practices
* when some parts may be a little more complex, it must be (1) very well documented, (2) provided with very easy way to use and understandable functionalities.
* Most of the people who will develop do not have a lot of experiences, so we can get regular feedback and identify areas where we have too much complexity

## Structure

In order to ensure security, and enforce every component to follow the same good practices, the files of components will not be directly accessible. Instead, only one entry point will be provided to the external world (/index.php), and every request will be processed by this entry point.

This entry point will analyze the request, and if it is correct, it will dispatch it to the right component after security checks.

Each component, located in its own directory (component/<component\_name>) will follow the structure below:

* page/<page\_name>.php
* static/<filename>
* service/<service\_name>.php
* locale/<language>.inc
* <component\_name>.inc

The *page* directory contains one file for each screen provided by this component. This file should have the extension *.php*, all other files in this directory must not have this extension.

The *static* directory will contain static resources (images, java scripts, css…) which are not dependent of the user using the application, and thus can be easily cached by web browsers or web proxies, even among different user sessions. Because we want to stay with simple design, we will not reach maximum performance, but to clearly distinguish those resources will help making the application more efficient.

The *service* directory will contain one file for each service provided by this component. A service is different from a page, in the sense that a page is in HTML format to be displayed in a browser, while a service may return data in any format. Those data are not intended to be displayed, but to be processed, and may be called either by a screen, or by a third-party system. A service is typically using XML as output, but is not restricted to it.

The *locale* directory will contain one file per language, each file containing the mapping between keys, and the corresponding localized string. This will be detailed in the next chapter.

The file <component\_name>.inc will:

* Ensure security checks to access to the pages and services: by default, nothing is accessible, and a component must implement the checks to allow access
* Define the DataBase elements it manages.
* Declare the components it is dependent on. Thus we will ensure there is no cyclic dependency, which would be the result of a wrong design.
* Store information about the user currently using the application, if needed (any attribute of the class will be saved in the session).
* Provide any other functionalities that may be accessed by other modules

More details are provided in the next section.

From outside, the files will be accessed using the URLs as follow:

|  |  |  |
| --- | --- | --- |
| **Type** | **URL** | **Will access in the structure to** |
| Page | /dynamic/<component>/page/<page>  /dynamic/<component>/sub\_page/<page> | /component/<component>/page/<page>.php |
| Service | /dynamic/<component>/service/<service> | /component/<component>/service/<service>.php |
| Resource | /static/<component>/<filename> | /component/<component>/static/<filename> |

The reason to separate dynamic elements from static elements into two different sub-directories from outside point of view is technical: to allow caching (no cookies reltive to the user’s session outside of /dynamic/).

The difference between URLs …/page/… and …/sub\_page/… is that a sub\_page means it is embedded (either in a frame, in a popup…) and so should not contain the usual header and footer, while a page will be automatically surrounded by the standard header and footer content.

On top of the components, is the class PNApplication: this is the class which instantiates and contains all the components. This class may be accessed from anywhere in order to access to a specific component. This class is also used to store what is the current domain (database) the user is using.

More details about the specification of a component is provided in chapter Component.

# 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 (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.

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. UserManagement component will have the namespace UserManagement). 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”…)

## 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.

To access 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*.

In addition, to make the build of SQL requests easier, but also to share among components the building of a final request, we will use a “SQL builder”, under the class *SQLQuery*.

Each component will declare its own part of the data model: the tables, columns, and access rights needed to access a specific table or column; but also the links between tables.

This way, it will make possible to design screens that can adapt to different situations or data models. This is detailed in the Component chapter.

## External libraries

External libraries, that may be used by any part of the application will also be located in the common part.

TODO: describe/put links to libraries used (To be done each time we add a new library)

# Component



## Pages and services

As explained before, a component can provide to the outside world *pages* (a screen, or a part of a screen that can be embedded by another screen), and *services* (to retrieve data, save data, or trigger functionalities).

Each page and each service must be protected, to ensure the user is allowed to see, or modify data: by default, the access is not allowed. The component must override *is\_page\_allowed* and *is\_service\_allowed* to implement security checks.

## Access rights

A component must also specify the access rights it will use to ensure security. More details about access rights are given in the chapter dedicated to the UserManagement component.

## Data Model

Each component specifies its own data model. A data model is composed of tables, each table containing typed fields (or columns). A table can be linked to another, through common fields. Each table and column can be restricted: to access to a table, or a column, a user must have the specified access rights.

Having the data model specified this way, will allow to implement generic functions to manipulate data, while ensuring that a user cannot access to a data if it does not have enough access rights. It will also allow to implement algorithm to know how data are linked together, thus for example when displaying a user with its username, we can easily propose to the user to display its first name, last name, … because we know that those data are linked to a particular user. Also when removing data, we can easily know which links are broken, and avoid keeping *unreachable* data in the database.

Finally, a table contains also a list of *displayable* columns. A displayable column correspond to data which can be displayed to the user, while other data are internal. For each displayable column, a localized string is specified.

Here is a partial representation of it:



# General Components



## Authentication

### Responsibilities

The Authentication component is responsible to validate a username together with a password.

As students and staff already have username and password in different systems, we will not define new usernames and new passwords again. Instead, we will use an external system, where people already have their username and password.

Depending on the project, the external system may be different (Active Directory in Cambodia, Linux in Philippines…). That’s why during authentication we will also provide with a *domain*, specifying on which project the user belongs to, and so which external system should we use to do the authentication.

In addition, the authentication system must provide the information if the authenticated user is a local staff or not.

### Functionalities



The authentication component will not manage any data, but only provide with the authentication system for a given domain.

An authentication system must implement an *authenticate* method, the others are optional.

When a user is successfully authenticated, the authentication system will return a string, corresponding to a token (or session ID) that we may use for subsequent operations.

Among the optional functions, at least the *get\_user\_list* method is recommended, to ease synchronization and avoid the need to create information manually. The others will allow to manage users directly in the software, but this is optional.

## User Management

### Responsibilities

The UserManagement component is responsible to maintain a list of known users (registered in external systems), to allow a user login and logout, and to manage access rights for the users.

The access rights define what a user can do or cannot in the application. Almost every page, service, data, or functionality must be protected by access rights.

Rights can be assigned directly to a user, or through roles. A role is defined by a set of access rights, and then roles can be assigned to users. A user can have several roles.

The total rights a user has is the union of (1) the rights directly attached to this user (2) the union of the rights of all the roles of this user. When the same right is present several times, the less restricted is kept.

Two particular roles will be defined by default:

* Local administrator: a user having this role has all rights
* Staff: in order to associate a set of rights for all staff from all projects. This will be particularly used when a staff from a project connects to another projects software or database.

### Functionalities



UserManagement depends on Authentication component in order to validate username and password when a user login.

The component will keep in the session information about the user currently authenticated:

* Its domain
* Its username
* The token returned by the authentication system when the user has been authenticated
* The result of the computation of all its rights, by domain (to reduce DataBase access, as any page will needs security checks, we will store the computation as soon as we need it)

The component will provide functionalities to other components:

* Login and logout
* Check if the user has a specific right, in the current domain

And 2 events the other components may subscribe:

* Logged\_in: raised when a user logged in, so other components may populate information about the logged user (like its first name and last name…)
* Logged\_out: raised when a user logged out.

UserManagement provides also classes for the other components to specify access rights:



Access rights are organized per category, in order to be able to display them in an organized way to the user. A category is simply a list of rights, with a localized name.

Each *Access Right* is named (unique name used to identify it), and has a translation so it can be displayed to the user.

We want to be as precise as possible in the rights assignments (meaning it is better to define several rights instead of one very general). For example: “can see the list of users”, “can see the rights of a users”, “can see the roles of a user”, “can edit the rights of a user”, “can edit the rights of a role”, “can assign roles to users”……

But we want to keep the security checks simple, to avoid mistakes, and we want the rights to be consistent. For this a right may imply other rights. For example: “can edit the rights of a user” implies “can see the rights of a user”, else it would be non-sense. That means that anyone having the right “can edit the rights of a user” gets automatically the right “can see the rights of a user”.

More generally, the implications specified are represented by a map: for a given value of the current right, it implies a list of rights together with the value of those rights.

### Data



A user is defined by its domain and username.

A user can have rights attached directly through the table *UserRights*.

A user can have roles attached through the table *UserRole*.

Each role is defined by a name (in table *Role*), and has rights attached through the table *RoleRights*.

Each right is defined by a name, and a value which may have any type (it will be the responsibility of the component managing this right to handle the value accordingly).

## Application

The component Application provides mainly the layout of the application, as well as the login page if the user is not yet authenticated. The layout provided will contain links to pages of other components, according to the access rights of the user.

It will also provide access to the different databases (different domains) to the user, so we can switch from one to another (by default when the user enter the application, it is connected to the local domain).

## DataList

*DataList* implements a generic screen, where the user can see a table of data, select or not the columns to display, make searches, edit data… (an Excel sheet++).

This screen is using the data model defined by the components, to know what data are available, and what the type of each data is.

Then, using this screen will need only few lines: give what is the starting point in the data model, and what are the data displayed by default. For example, if we want a screen with a list of users, the starting point is the table of Users, then we will be able to see any data having relationship with a user (its personal information, through the component *People*, …) just by analyzing the data model.

To perform that, the DataList will analyze the data model, by starting from the given starting table, and trying to go through all links and foreign keys to reach data. Thus it will know what are the reachable data. While going through the data model, it will always check that the user has access to tables and columns. If for example a table is not accessible, it will not try to use it to reach data, even if this is just a step to reach a data in another table which is accessible.

The component provides a service *get\_data*, used to retrieve data asynchronously using AJAX requests.

An example of its usage is provided below:

$list = **new** DataList("Users");

$list->primary\_key("Users.domain","Users.username");

$list->add("Users.domain", **false**);

$list->add("Users.username", **false**);

$list->add("UserPeople.people>first\_name", **false**);

$list->add("UserPeople.people>last\_name", **false**);

$list->add("UserRole.role\_id>name", **false**);

In this example, we want to display a list of users. We first indicate what is the *starting table* “Users”. Then we indicate what are the primary keys (those fields will always be displayed, and can be used to identify uniquely an entry).

Then we can specify data that will be displayed by default to the user. For this, two forms are available:

* *Table.column* to display the data of the given column from the given table
* *Table1.column1>foreign\_column* : this form is used to follow links between tables. For example *UserPeople.people>first\_name* means that the data to display is the first name, and to access it we have first to use the table *UserPeople* which contains a foreign key *people*. This foreign key points to another table, *People*, which contains a column *first\_name*.

It is possible to use several foreign keys like key1>key2>key3>finalcolumn.

As we can see, it is not necessary to start with the *starting table*. In our example, the starting table is “Users”, but we can specify data to display, starting from *UserPeople* or *UserRole*. In this case, the DataList component will find automatically how to reach this table, by using the specification of the Data Model.

In our example, we can replace *UserPeople.people>first\_name* directly by *People.first\_name*, and Data List will still find the path to reach the first name.

Thus, DataList provides an easy way to build a screen containing a list of data, while ensuring security, because DataList will always check that the current user can access to every table and every column before to use them.

## People

### Responsibilities

### Functionalities

### Data



## Storage

### Responsibilities

### Functionalities

### Data



## Contact

### Responsibilities

### Functionalities

### Data



# Selection Components

# Education Components

# Training Components

# External Relations Components

# Finance Components