**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 Title 1 3](#_Toc307307689)

[1.1 Title 2 3](#_Toc307307690)

[1.1.1 Title 3 3](#_Toc307307691)

# 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

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 exist (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 to use and understand functionalities.

## Structure

In order to ensure security, and enforce every component follow the same good practices, the files of components will not be directly accessible. Instead, only one entry point will be provided (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 is 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. To clearly distinguish those resources will then help making the application more performant.

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 display in a browser, while a service may return data in any format, is not intended to be displayed, and may be called either by a screen, or by a third-party system. A service is typically using XML as output.

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.
* Provide any other functionalities that may be accessed by other modules

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 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 instantiate and contain all the components. This class may be accessed from anywhere in order to access to a specific component.

# Common

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

## Localization

In order to make the application multi-language, any data 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 exemple, 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 requested string.

The language used is kept by in the session of the user, and 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*, 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 define by other components.

## 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, by using *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, because we know what is the data model and relations between data, and this in an efficient way as we don’t need to access the DataBase.

Finally, *DataList* implements a generic screen, where the user can see a table of data, select or not the column to display, make searches, edit data… This screen is using the data model defined by the components, to know what data are available, and what is the type of each data.

Then, using this screen will need only few lines, to give what is the starting point in the data model. 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 attach any data having relationship with a user (its personal information, through the component *People*, …)

# 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.

### 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. At least the *get\_user\_list* method is also recommended, to ease synchronization and avoid the need to create information manually. The other methods are optional, if they are provided it means more functionalities will be available and users can be directly manage from the application.

## User Management

### Responsibilities

The UserManagement component is responsible 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 assign directly to a user, or through roles. A role is defined by a set of access rights, 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.

### Functionalities



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

The component will keep information about the user currently authenticated:

* Its domain
* Its username
* The result of the computation of all its rights

The component will provide functionalities to other components:

* Login and logout
* Check if the user has a specific right

And 2 events the other components may subscrive:

* 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.

### 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, 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).