**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

## Overview

The application is separated into *components*, each component being responsible of specific functionalities and data.

A component can provides different resources:

* Dynamic pages: web pages which can be displayed to the user, and being dynamically generated (according to the user, its permissions, content from database…)
* Static resources: resources which are not dependent of a user, or database (typically an image, css file, a javascript…)
* Services: a service is also dynamic (depends on the user, its permissions, do actions on database…), but the difference with a dynamic page is a service is not supposed to provide something to display to the user. A service is typically used to manipulate data: add a new information in database, retrieve data, modify data…
* Translations: in order to make the application multi-languages, each word or sentence can be provided in different languages.

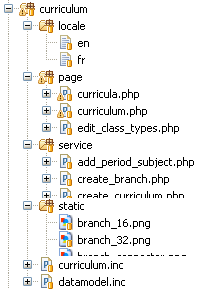
A component can also provide functionalities to other components, to be used directly in PHP and not exposed to the client.

The application also contains a *common* part: some resources (PHP, JavaScript, images…) which are not specific to a functional area, but are aimed to be shared and used by any part of the application. For example, it contains common images, so every page can use the same images for common purposes, or some UI widgets, which will make the application more homogeneous. Also some functionalities like access to the database, or management of translations.

## Structure of a component and URL to access a component

Each component must follow the same structure: each kind of resources must be located in a specific directory, and will be accessed through a common URL structure.

Here is an example if a component *curriculum*:



Component implementation and data model specification

Static resources

URL to access:

/static/curriculum/branch\_16.png

URL to access:

/dynamic/curriculum/service/create\_branch

Services

URL to access:

/dynamic/curriculum/page/curricula

URL to access:

/locale/curriculum/

Dynamic pages

Localized strings: translations in English (file *en*) and French (file *fr*)

### Dynamic pages

Every dynamic page must be located in directory *page* of the component, and have the extension *php*. The URL to access it has the format /dynamic/<component\_name>/page/<page\_name>. For example, the URL /dynamic/curriculum/page/curricula will access to the file curriculum/page/curricula.php

### Services

In the same way, services must be located in directory *service* of the component, and have the extension *php*. It is accessed through URL format /dynamic/<component\_name>/service/<service\_name>

### Static resources

Static resources must be located in directory *static* of the component, and are accessed through the URL format /static/<component\_name>/<filename>

### Localized strings

Translations of words or sentences used by the component must be located in the directory *locale* of the component. For each language, a file named with the language code will contain the translations (example, for English, the language code is *en*). More details about translations will be provided in next sections of this document.

### Component implementation

A component must provide a class implementing the abstract class *Component*.



This class will mainly allow to access to pages and services, implementing security checks, and provide the list of permissions (rights) the component supports. This class will also allow to provide functionalities to other components.

More details will be provided in the next sections of this document about PNApplication and Component classes.

### Data model specification

Each component must declare its data model (tables, format of columns, links between tables, permissions needed to access to tables or columns…). This will be used, first to ensure security checks so we can be sure that if a user should not access to specific data, even a component *forget* to check the permission, the access will be automatically denied. It will also be used to implement generic functionalities to manipulate data. More details about data model will be provided in the next sections of this document.

## URL explanation

### Static and dynamic

The reason to separate clearly dynamic and static resources into URLs starting either with /dynamic/ or /static/ is for performance purpose for static resources.

Indeed, static resources should be cached by the browser of the user: the first time, the browser will download the resource, and then will store it so it does not need to download it again the next time it needs it.

For a resource to be cached by the browser, there are several conditions: the resource must be marked as cacheable, and the request to access it must be exactly the same.

As we are using PHP, with sessions, by default PHP will add a cookie containing the session ID, that the browser will include in every request to the server. That means that every time the same user connect to the application, it will get a different session ID, so the request to a static resource will be different, avoiding the browser to reuse the resource previously downloaded.

Cookies may be restricted to a given path: in our application, we will specify that the cookie containing the PHP session ID is restricted to the path /dynamic/, it means every request to an URL starting with /dynamic/ will contain the session ID, but any other request won’t. So the URLs starting with /static/ will not contain any cookie, thus may be cached.

### Single entry point

In order to ensure security checks, and that we will not allow to access to a file which is not supposed to be sent to the browser, all requests are filtered: the web server is configured to redirect all the requests to a single entry point: index.php

This PHP file will analyze the requested URL, process it, and reject any restricted resource or unexpected request.

## Security for pages and services

Every component must implement security checks. In order to make this sure, any page or service is by default restricted. Thus, a component must explicitly allow its access, after needed security checks.

When the URL /dynamic/selection/page/dashboard is received by the server, here are the steps:



1. The application receive the URL “/dynamic/selection/page/dashboard”. It is analyzed by the single entry point “index.php”: this is a request for page *dashboard* of component *selection*.
2. The component *selection* is retrieved through the *PNApplication* object.
3. The page *dashboard* is requested to the component.
4. The component checks the access to this page is allowed. If not the request is rejected.
5. An object *Page* is created, and requested to generate the page.
6. The file *dashboard.php* located in directory *page* of component *selection* is executed.

Details about the class Page is provided in the next sections of this document.

A similar process is used for services.

Any static resource is allowed: if a resource is restricted, it means it depends on the user, so this is not static anymore. Same for translations, there is no need to secure access to translations.

## Security of data

In order to ensure security of data, direct access to database from a component must be avoided. To access to the database, a PHP script must use one of the classes provided in the *common* part (SQLQuery or DataModel), which will ensure security checks.

Each component declares its part of the data model in a file *datamodel.inc*. The data model is declared using the class DataModel, allowing to:

* Declare tables
* For each table
  + Declare columns, with name and type
  + Specify access restrictions to the table: who can access the table, can add or remove entries in the table…
  + Specify restrictions on each column: who can access to data on this column, who can change a data…

Then, typical way to access to the database is using SQLQuery: this class provides functionalities to easily create SQL queries, and will do all necessary security checks before to send it to the database.

More details on those classes are provided in next sections of this document.

# 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”…)



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, as mentioned in the chapter “Security of data”, 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*.

As mentioned before, each component declares its own part of the data model. This is done through the class DataModel:



### DataModel class



The DataModel class contains the list of tables.

When specifying the data model, a component use the method addTable to declare a new table.

When accessing the model, the methods getTable and getTables are used. Those 2 methods perform security check: getTable throws an exception in case the requested table is protected and should not be accessed by the current user. getTables only returns the tables which are accessible by the current user.

### Table class

The Table class represents a table in the database. It contains a list of columns, links to other tables, list of indexes, and the specification of permissions to access to the table and its columns.

A column is represented by the abstract class Column. Every implementation of this class represents a type of data which can be stored in the column.

Every column has a name, can\_be\_null indicates if the null value can be used or if we must give a value (column is optional or not), and unique indicate if the value of this column must be unique or not in the table (no duplicate allowed).

When a ForeignKey is used, we automatically know that there is a link to another table. However, a table can be linked to another without using a foreign key, in this case the class Link is used to indicate the link.

Different kind of permissions can be set on a table and its column:

* read access for the entire table
* write access for the entire table
* read access to specific rows of the table
* permission to add a new row in the table
* permission to remove a row in the table
* for each column, read and write access.

The accesses are specified by indicating the rights the user must have to get this access. The management of rights a user has will be discussed in the section of component user\_management.

### SQLQuery

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*.

When the SQL query is ready, before its execution it will use the data model to check the permissions, and then ensure the current user has the right to access to the data implied in this SQL query.

## 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…

# 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.

### 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 rights/permissions 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.

One particular role will be defined by default: Local administrator. A user having this role has all the rights and can do everything. This role should not be assigned to a normal user, but to a special dedicated user which will be used only when needed. This to avoid doing mistakes: if someone has all the rights each time he/she connects to the application, this may lead to mistakes.

### 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).

### Access rights

The UserManagement component itself defines rights:

#### Read rights

* consult\_user\_list (Boolean): if true the user can see the list of users
* consult\_user\_roles (Boolean): if true the user can see the roles attached to the users
* consult\_user\_rights (Boolean): if true the user can see the rights another user has
  + Implies consult\_user\_roles = true

#### Write rights

* manage\_roles (Boolean): if true, the user can create, remove and edit roles
* assign\_roles (Boolean): if true, the user can assign or unassign roles to users
  + implies consult\_user\_list = true
  + implies consult\_user\_roles = true
* edit\_user\_rights (Boolean): if true, the user can assign or unassign rights directly to a user (without using a role)
  + implies consult\_user\_list = true
  + implies consult\_user\_rights = true

## 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).

This component also provide a “search page” allowing the user to perform searches on the database.

Regarding the layout, the component provides also 2 classes to layout sub-pages: MainSectionPage and SubPageHeader.

## Data\_model

The data\_model component provides the classes for other components to declare their part of the data model: see chapter “DataBase” above.

Its goal is also to provide generic functionalities, pages (or part of pages) and services for the data model.

### DataModel class

The DataModel class, together with Table class are described in the chapter “2.2 DataBase”: they allow each component to declare their part of the data model, building the complete data model of the application.

In addition to what has been previously described, *displayable data* can be declared on each table. A displayable data is a column of a table which aims at being displayed to the user (contrariwise the other columns are internal data, like foreign keys, which are not supposed to be displayed to the user). When declaring a displayable data, a localized name is provided, so the description of the data can be displayed.

Those displayable data will allow to implement generic pages, which can display data from one or several tables: the page will automatically adapt to the data model, by analyzing it. Thus a modification of the data model will be automatically reflected in those pages, without any additional code.

#### Declaration from each component to build a complete data model

Each component declares its part of the data model, in a file *datamodel.inc*.

The class *DataModel* will then execute each of those files, to finally get the complete data model.

#### Specific data for a project

Each PN project may have specific needs for the data. For example, in Cambodia, we would like to store the name of a person in English, but also its Khmer version. This is obviously not needed for Philippines or Vietnam.

In order to keep the databases of all project homogeneous, so all databases will have the same structure, some data may be hidden for some projects.

For this, a component may provide a file *datamodel\_hidden.inc*: it declares, for each project, which data should be hidden. Thus, the Khmer name column will be present in the database of each project, but will never be displayed when we are connected to the Philippines or Vietnam application.

This will allow flexibility for each project to have data which has not meaning or no interest for other projects, while keeping homogeneous database structures among the different projects.

### DataPath and DataPathBuilder

The class DataPathBuilder can automatically find the path from one table to another, or find all possible linked data from a given table, by analyzing the data model.

### Pages

#### Entity

A portion of page is provided to display data from a given row of a table. Every displayable data of this table is displayed. If the current used has the permission to modify a data, the page will provide this possibility to the user: the data will be locked when the user starts to edit, to avoid another user attempt to edit the same data at the same time.

Each data will be displayed according to the type of the column: for example, if the column is a date, and the user wants to edit the data, a date picker (with a calendar) will be displayed so the user can easily pick a new date.

#### Editable entity list

This page allows the user to edit a list of rows, add new rows, or remove rows.

This can be used typically when a table contains a list of entries, like a list of provinces in a country, so the user can easily edit this list.

In the same way as *Entity*, each data will be displayed/edited according to the type of the column. For instance, if a column is an integer, the user will be able only to enter digits.

#### Data list

*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… (a kind of 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. For this, DataList is using DataPathBuilder to know all the possible data which can be linked to the starting table, and how to reach all those data (following links, foreign keys…).

An example of usage, for a list of user:

$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**);

Here, with only few lines, we indicate:

* The starting point is the table “Users”
* To uniquely identify a row (a user), we have to use the columns “domain” and “username” of the table “Users” (primary key)
* Then, by default, the table will contain the following information: “domain” from table “Users”; “username” from table “Users”; “first\_name” and “last\_name” from table “People” which can be reaches through “UserPeople”; “name” from table “Role” which can be reached through “UserRole” table.

The model being:



So for example, we can reach “first\_name” through the table “UserPeople” which is linked to the starting table “Users”.

### Services

Services, used by the previous pages or functionalities, are provided to:

* Lock a row in a table, or an entire table
* Release a lock
* Retrieve data for *DataList* (with columns selected by the user)
* Export data from *DataList* in different formats like Excel or PDF.

## Storage

### Responsibilities

### Functionalities

### Data



## Geography

### Responsibilities

Geographic information will be centralized in a specific database. This component will be responsible to store information like country administrative divisions (states, provinces, regions…), cities, cities sub-divisions…

It will also provides with localized names of each part (country name, city name…).

Ideally, this database should be populated from data retrieved from official sources, or web sites providing those kind of data (like geonames.org).

It will also ease the integration with map services (like Google Map), if needed.

### Data



A country can be divided into up to 3 levels of administrative divisions (regions, provinces…), then into cities. Each city can also contains divisions (like Barangay in Philippines, arrondissement in France…).

*CountryDivisions* allows to specify which levels of division are used, and their name (for example, in USA the first level corresponds to a state, while in another country it may be a province, or a region…).

# Contact

## Responsibilities

The Contact component is responsible to keep a database of contacts.

A contact is represented by a *ContactEntity*. Each contact can be labeled by a type (high school, IT company, NGO…).

A contact may have postal addresses attached, as well as contacts like emails, phones, Instant-Messaging…

The centralization of those data will ease searches, to have homogenous database for this kind of information, as well as no duplication.

## Functionalities

## Data



# People

## Responsibilities

The People component is responsible to keep general information about a physical person.

Using this component, every other component will be able to store information about a person (which can be, depending of the component, a student, or a staff, or a contact in a company, …).

The advantage of having this in a dedicated component will be to homogenize those kind of information, but also to centralize it and avoid duplication of data about a physical person.

This way we will ensure to always have similar information about a person, easing searches, or reuse of pages, but also the possibility that someone who was a student, become a staff: this is still the same physical person, so information about this physical person remain unchanged and centralized, while we can make many links to this person, to attach roles, jobs…

## Functionalities

## Data

