# MANAGING USERS AND SERVICES USING AN LDAP BASED WEB-APPLICATION

C. Bouras[1, 2] V.-J. Khan[1] A. Limperis[1] C. Sintoris[1,3]
[1] Computer Technology Institute-CTI, Riga Feraiou 61, 26221 Patras, Greece
[2] Department of Computer Engineering and Informatics, University of Patras, 26500 Rion, Patras, Greece
[3] Department of Electrical and Computer Engineering, University of Patras, 26500 Rion, Patras, Greece
E-mail: bouras@cti.gr

## ABSTRACT

This paper describes the design and implementation of an open and adaptive environment for managing LDAP databases. Our work was mainly focused on the administrative needs of the Greek Schools Network. Albeit we applied considerable effort in order to construct an application that would not be bound on limitations arisen by the specific characteristics of the Greek Schools Network. Such limitations are the directory technology used and the directory schema applied. The result of this effort is a design which is extensible, adaptive to a great variety of LDAP structures and which provides the necessary flexibility to perform even the most demanding administrative tasks.

**KEY WORDS**
LDAP, php, web-application, DIT, schema

## 1. INTRODUCTION

Managing a large number of users, who should have access on a variety of services over a wide area, can become a difficult task. In our case, the Greek School Network (from now on referred to as GSN [1]), with some 10,000 users, it is even more difficult, since there is no central managing authority. The GSN connects all schools and educational administration units in Greece with each other and provides them access to internet. The networking infrastructure is used in order to provide internet connectivity to school labs or email accounts to teachers and educational personnel.

The GSN is split into a small number of responsibility areas over which a different authority has control. On the other hand, efforts were made to keep the network's structure uniform, in order to allow tasks such as maintenance and upgrading to be more easily performed. The users of the GSN should have access to services like email, web space and dialup. Passing all the services through an LDAP database provides a clean and relatively easy to manage solution. All relevant data is kept in one place and changes to the profiles of the users don't harm consistency. The problem that arises is how to effectively manage the LDAP database itself, since many different types of users should have access on it. Anonymous users should be able to apply for an account, users should be able to change some of their data and various levels of administrators should have a different level of access on the LDAP database and all that through a user friendly interface.

Our first attempt, the User Management Environment (from now on referred to as UME), already provided a solution. The UME was a web based LDAP administration application which met exactly the needs of the GSN. It was running stable for over a year, servicing administrative tasks like creating, modifying, deleting and searching for user accounts. The UME's strength proved to be its weakness, since it was coded just to fit the needs of the GSN. Extensions and alternations of the GSN's structure, even minimal ones, require the recoding of parts of the UME. In order to overcome just the extensibility limitations of the UME, we began around June 2001 to design and implement a new web based environment. Our goal was to create an environment which would have as less requirements as possible on existing structures.

The amount and variety of the users as well as that of the services provided to them by the GSN, pointed to an implementation that would not be bound to narrow specifications. Services already provided to the users of the GSN are constantly evolving and new services will have to be added as new user demands are arising. In order to cope with the ever changing nature of the GSN, we chose to design a solution that could adapt to post-defined demands. By aiming to meet those requirements, we succeeded in designing a medium that is not only suitable for the GSN, but can also satisfy a very wide and diverse set of directory related tasks.

Before starting designing our own solution, we surveyed a number of existing LDAP interfaces. Our target was to find one which we could easily alter in order to fit our demands, without having to invest inappropriate amounts of time and effort. Although many of the projects were quite mature, none of them actually met our demands. Below we are reviewing some the applications that already exist.

1. LDAP Admin [2]

LDAP Admin is a mere web interface of Microsoft's Outlook Express Address Book, which can work along with the specific product. It is very limited though, because it manages only the address book and nothing else.

2. LDAP Explorer [3]
   It is a simple web-client for LDAP Directory servers developed with PHP, but it does not have any advanced features.
3. web2ldap [4]
   It is an LDAP web interface in Python. Although it is in a mature state, its main disadvantages are that it does not provide any functionalities apart from the basic LDAP features such as search, rename, add, delete, etc. and its user interface is oriented to experienced LDAP users.
4. Likken [5]
   Likken is a JSP based LDAP client still in an early stage. It offers only the basic functionality of viewing / editing entries.

Most of the above applications provide just a web interface which allows the performance of only basic tasks. Those tasks are specifically attached to the LDAP philosophy, meaning that a user of those applications is required to be quite experienced in order to operate them. Instead, our design aims in hiding the details of the LDAP architecture allowing the user to operate in a per task manner.

## 2. GREEK SCHOOLS NETWORK

The GSN's basic aim is to interconnect schooling facilities, personnel and students with each other as well as to provide them access to the internet. The structure of the GSN is divided into three major sectors:

- Backbone network
  As backbone network the GSN uses infrastructure provided by the Greek Research & Technology Network [6] (GRNET), which composes the most developed network in Greece.
- Distribution network
  The distribution network is organized in two layers, the first one consisting of 9 wide geographical areas and the second one consisting of 51 smaller areas.
- Access network
  The access network connects each school or administrative office to its familiar prefectural node.

The entities of the GSN that have to be managed through the LDAP database are: Root dn, Prefecture, School unit or Administrative unit, Unit account, Teacher account, Student Account, Personnel Account, Group of users.

The LDAP DIT used to store the above entities is a hybrid between hierarchical and flat structure. The form of the hierarchical part is `<unit>`, `<prefecture>`, `<country>`. User and unit information is stored in a flat manner under `ou=people, <country>`. Group information can be stored everywhere in the DIT. There are also some special purpose structures, such as the `<deleted>` entry. That keeps track of recently deleted accounts in order to prevent the use of recently deleted email addresses, uids etc. The `<new-entries>` entry stores information about recently added accounts in order to inform administrators.

## 3. DESIGN ISSUES

Before deciding about which LDAP server we would use, we tested and did large parts of the implementation both on OpenLDAP [7],[13] and the iPlanet Directory Server [8],[13]. About 50% of the implementation took place on OpenLDAP, before switching to iPlanet. The decision about the backend directory server was based mainly on licensing aspects.

Apart from the existence and configuration of the backend components, there are a number of additional steps needed to be performed in order to get the system up and running. They basically consist of a number of LDAP entries which define data such as the types of objects that will be used, the types of users that will have access to the system, the actions allowed to be performed on the directory entries etc. All that information is stored under `ou=Config, cn=en`, where 'cn=en' denotes the English language. The entities that exist under `ou=Config, cn=en,` are `cn=Objects`, `cn=matchtypes`, `cn=datatypes`, `cn=UserLevels`, `cn=SpecialActions` and `cn=Reports`.

- `cn=Objects` contains information about the entities that are to be kept track of: Teachers, Students, Accounts and Prefectures etc. Each entity contains information about its parent object, meaning an entity of that type can hang only under an entity whose type is the same as that defined by the parent object (`umdParentObject`) attribute. For example, the entry `cn=teacher, cn=Objects, cn=en, ou=Config, <base dn>`, as shown in Figure 1, contains following attributes:

| Attribute | Value |
|---|---|
| cn | teacher |
| description | Teacher |
| objectClass | top |
| objectClass | extensibleObject |
| umdForms | Teacher_Anonymous:Anonymous |
| umdForms | Teacher_Root:Admins |
| umdForms | Teacher_Root:Root |
| umdForms | Teacher_User:Users |
| umdParentObject | School |
| umdParentObject | AdminUnit |

*Figure 1 - The entry for cn=teacher, cn=Objects, cn=en, ou=Config*

The umdForms attribute defines which form should be used to view, modify or add an entity with the attribute umdObject=teacher.

- cn=matchtypes describes the filters that can be used in searches. As one can see in Figure 2, this is being defined by the multivalue attribute umdValue. The information contained in that attribute has the form "name of the matchtype <tab> definition".

| Attribute | Value |
|---|---|
| cn | matchtypes |
| objectClass | umdValueItem |
| objectClass | top |
| umdValue | "Different from" "(!(%a=%v))" |
| umdValue | "Does not include" "(!(%a=*%v*))" |
| umdValue | "Sounds like" "(%a~=%v)" |
| umdValue | "Is" "(%a=%v)" |
| umdValue | "Includes" "(%a=*%v*)" |
| umdValue | "Ends with" "(%a=*%v)" |
| umdValue | "Starts with" "(%a=%v*)" |
| umdValueDescription | Matchtypes |

*Figure 2 - The entry for cn=matchtypes, cn=en, ou=Config*

- cn=datatypes is used in order to create the appropriate search box for each type of entry. Figure 3 shows the contents of the datatype entry for the entry type teacher. The umdInterestingAttribute attribute contains information about which attributes of an entry of the type teacher are searchable and the labels that should be displayed. Also, there are attributes that define the search scope, the base dn and the filter of the search.

| Attribute | Value |
|---|---|
| cn | Teachers |
| displayName | Teacher |
| objectClass | extensibleObject |
| objectClass | top |
| umdInterestingAttribute | "cn" "Full name" |
| umdInterestingAttribute | "givenname" "Name" |
| umdInterestingAttribute | "sn" "Surname" |
| umdInterestingAttribute | "uid" "Userid" |
| umdInterestingAttribute | "accountstatus" "Account status" values(active,Active,disabled,Disabled) |
| umdValue | user |
| umdValueDescription | Object category |
| umdbasedn | ou=people,dc=sch,dc=gr |
| umdfilter | "(umdObject=teacher)" |
| umdscope | SUBTREE |

*Figure 3 - The entry for cn=datatypes, cn=en, ou=Config*

- cn=UserLevels is used to define user levels. The umdValue attribute is used to define user levels and their names, separated by an ":".
- cn=SpecialActions can be used to define some non standard actions. For example, the custom action "Activate", defines through the umdValue attribute that the attribute accountStatus of an entry of the type 'user' or 'teacher' should take the value active after performing that action.
- cn=Reports is used in order to create printable reports about the entries.

As one can easily derive from the preceding statements, the whole concept has no limitative dependence on existing directory schemas. The only attribute needed to be added to an existing entry is one that defines the type of that entry. Apart from that, in order to get the application working, one also needs to configure the above options and to create the appropriate forms. We will examine the forms concept later.

Group definitions essentially are entries which have their umdObject attribute set to "Group". The multivalued attribute named member contains the dn of each member. Also, each member entry carries a multivalued attribute named memberof which contains the dn(s) of the group entries the member belongs to.

For example the group ou=teachers, ou=prefecture, <base dn> contains the attribute 'member' with the value "ou=teacher1, ou=people, <base dn>", which is the dn of a 'teacher' entry. Likewise, the teacher1 entry contains the attribute memberof with the value "ou=teachers, ou=prefecture, <base_dn>". So group information about teacher1 is stored in two places.

Although it may seem that this approach creates overhead when adding or removing members from groups, it allows the efficient retrieval of grouping information: finding out the members of a group or the groups a member belongs to becomes a fast and easy task.

## 4. IMPLEMENTATION ISSUES

The basic concept of the application is the clear distinction between two main functions: the browsing and the performing of actions. These two parts are completely distinct and they do not interfere with each other (*Figure 4*). The browsing module does not depend at all on the actions module, while the actions module only needs a dn, which it gets from the browser or the search module, and the type of action in order to operate. The user can browse through any objects of the database and trigger actions on them at will.

The interface of the application consists of two frames, the browser (or the search module) and the action screen. The browser screen provides a collapsible tree-structured look on the directory server's entries, a group view of these entries and a search facility. By selecting an entry from the browser screen the action screen is activated and provides a set of possible and, depending on the access level of the user, allowed actions on the selected entry. These actions can be

summed up to a number of four: “view”, “modify”, “add”, “delete”. Apart from them one can add more actions. For that he would have to actually write code, since the logic of any new action has to be defined. For example, in the case of the GSN we have added an “activation”-action for newly registered users. The “activation”-action changes the value of an attribute of a user entry.

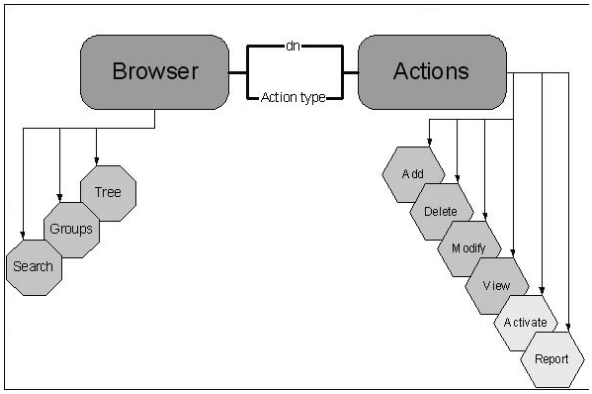


*Figure 4. Main Architecture*

Based on the information received from the browser module, the action module launches the appropriate action. For the actions “View”, “Modify” and “Add” an appropriate form is loaded. For the “Delete” action the form consists of a simple confirmation dialog. The forms are chosen using the dn of the object received from the browser module. There are distinct forms for distinct entry types only, not for distinct actions. That means that for adding, viewing or modifying an entry of the type “teacher” a different view of the same form is launched.

In order to have that adapting system of forms to perform the available actions, we implemented a form builder which provides forms in a manner of entry type/user type.

There is only one form definition for each entry type/user type pair. All actions on these pairs are performed using a different view on the same form. This is possible by using some flag values on the form definition. For example, if the attribute “`user_id`” should be viewable but not modifiable, the flag `viewable_user_id` is set to 1, while the flag `editable_user_id` is set to 0.

Bulk actions are actions that can simultaneously be performed for a large number of entries. Using this facility one can perform an action on multiple entries. Using the results of a search, one can delete or activate (which is a custom action) multiple entries.

Bulk adding can be performed in a different manner. The user first has to choose the type of object he wishes to add, then either upload a tab delimited file or enter the data manually in a text area field.

Security is implemented using access lists (ACL’s) that are defined on the directory server’s side. On the side of the application, by checking for group membership, the user is given a different view of the directory. For example, the administrators of a prefecture will see the area of the sub tree that is underneath the prefecture entry and nothing else. Of course, using the above procedure, a more detailed access policy can be defined.

## 5. FUNCTIONALITY

Whenever we trigger an LDAP-search to the directory adding the filter: "`(objectClass=*)`" [9] we are sure that the directory server will return all the entries that it has stored.

Starting off from the root of the directory, we trigger an one level LDAP search and so we are in the position to create the first branch of the browser tree from the results returned by the search function.

The links that represent the nodes and leafs of the tree display an attribute, which describes in a user friendly way the specific entry of the directory, e.g. the description attribute (`description`), or the canonical name (`cn`), or the user id (`uid`). This attribute is chosen by the browser module in the following way: The module creates a list of interesting attributes by reading the ‘`umdInterestingAttribute`’ attribute, as defined in the object type definition of the entry (Figure 3), and displays the first of them found. If an attribute is unavailable, it proceeds to the next etc. If none is found, then the dn is displayed.

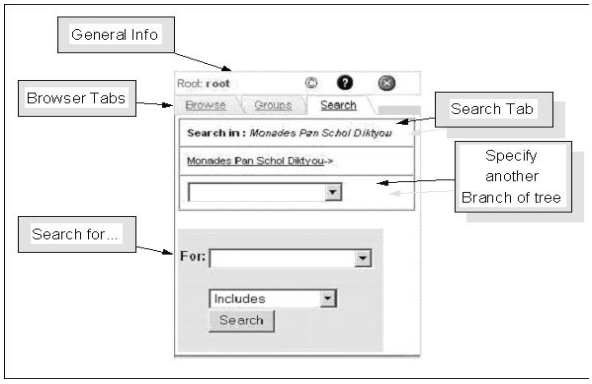


*Figure 5: Search feature*

Search functionality is provided using a drop down menu with which the user denotes the kind of object he is searching for e.g. teacher, student, prefecture, school etc (Figure 5).

In the next step he has the option to fill-in one or more of several fields (Figure 6), which are created dynamically, depending on his choice in the previous step. This is being done by reading the `cn=<umdObject>, cn=datatypes` configuration entry, which describes which attributes are searchable for each object type (Figure 3). For example, entries of the type teacher are searchable by using the "`uid`" or the "`cn`" or the "`sn`" field or any combination of those.

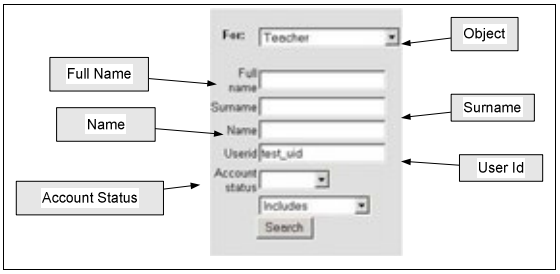


*Figure 6: Step two - filling search parameters in a field*

Also, he can use matching filters, like "`starts with`", "`contains`", "`ends with`", "`is`", "`is not`", "`sounds like`" etc (Figure 7). These matching filters are again defined in a configuration entry, under `cn=matchtypes`. (Figure 2)

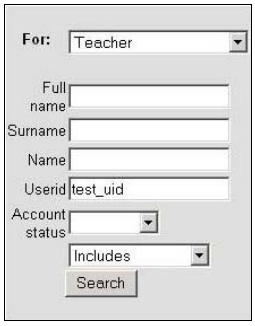


*Figure 7: Use matching filters*

After this step, the application creates the proper filter and conducts an LDAP search with a sub-tree scope. In order to prevent overhead by bad filters which cause an excessive return of entries, there is an upper limit of returned results.

Actions on directory entries are performed through forms. These forms are launched depending on the entry type and the user type.

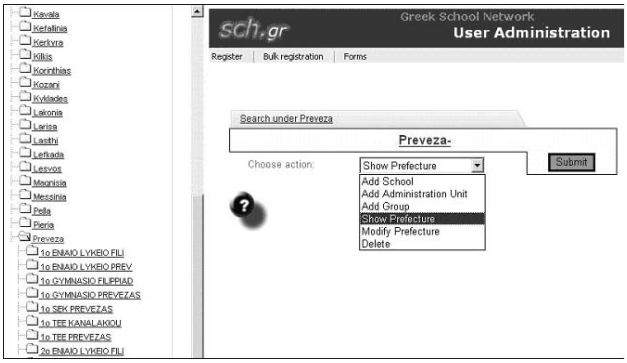


*Figure 8: Single Action*

There are several actions that can be performed on a single (Figure 8) or multiple (bulk actions) entries at a time. The available actions are Add, Modify, View and Delete. These actions depend on the value of the `umdObject` attribute of the entry. That value defines which form will have to be used on that particular entry.

The form builder part of our application can be used to create appropriate forms for each user type – entry type pair. Each form created can perform add, modify or view actions. View actions just display a form which shows the elements marked by the form builder as "`viewable`" in read only mode. Modify actions do the same, except for the elements marked as "`editable`", which allow the user to change their current values. Add actions are empty forms which are used to create new entries. The elements that are defined as "`calculated`" show only a "`calculated`" label. In all of the three actions the elements defined as "`hidden`" are never displayed.

The form builder consists of three steps:

1. Select the attributes
   Either by parsing `.schema` files or by querying a version 3 LDAP server, the form builder reads object and attribute definitions.
2. Configure the attributes
   In this step various options can be configured for the attributes selected in the previous step. Such options include labels, control types (input box, radio button, checkbox etc.), control options, calculation formula etc.
   When finished with the definition of these configuration options the user can proceed to the visual arrangement of the attributes on a canvas.
3. Arrange the configured attributes visually on a canvas.
   This step allows the user to visually design the form. An empty canvas appears on the middle of the screen, on which the user can drag and drop the fields defined in step 2.

During the whole process the user can move back and forth, refining configuration options or clearing out mistakes.
Using a load dialog, the user can load a previously stored form, reconfigure every option available and store it under a different name.

By changing a configuration option of the application, the whole interface can display itself in a different language, provided of course that the appropriate language definitions exist.

## 6. FUTURE WORK AND CONCLUSIONS

One can confidently say that our efforts to build an environment for managing users and services over LDAP succeeded in meeting the demands of an open and extensible application as well as in filling a gap. Looking back, we would surely revise some parts of the application in the way that they were developed, but we consider the overall strategy successful.

After reaching our goals, we started considering future enhancements. It is in our intentions to design and implement the following features:

- Rollback mechanism. This feature should provide more than basic rollback functionality. A major part should be implemented using backend procedures.
- LDAP Controls [10]. Some parts of the application can be rewritten in order to make use of LDAP Controls such as the Server-Side Sorting Control [11] or the Virtual List View Control. Albeit, the current release of the php LDAP libraries does not allow the use of this features.
- Library enhancements. During some development stages we encountered some limitation of the current php LDAP interface implementation. If the features we require will not be implemented in the near, we intend to write some php extensions ourselves.
- Open Source. Finally, we intend to release the environment to the Open Source community, as soon as some minor modifications are completed.

## REFERENCES


[1] The Greek School Network, http://www.sch.gr/index_en.php
[2] LDAP Admin, http://www.savoirfairelinux.com/download/ldapadmin.html
[3] LDAP Explorer, http://igloo.its.unimelb.edu.au/LDAPExplorer/
[4] web2ldap, http://web2ldap.de
[5] Likken, http://www.likken.org
[6] Greek Research and Technology Network, http://www.grnet.gr/index_en.html
[7] OpenLDAP Project, http://www.openldap.net
[8] iPlanet Directory Server, http://www.iplanet.com
[9] Timothy Howes, Mark Smith and Gordon Good, *Understanding and deploying LDAP directory Services* (Macmillan Network Architecture and Development Series)
[10] *LDAP SDK for C Programmer's Guide* (iPlanet™ Directory Server Resource Kit, June 2001)
[11] T. H. Loudcloud, A. Anatha, *RFC 2891: LDAP Control Extension for Server Side Sorting of Search Result* (August 2000)
[12] iPlanet Directory Server Documentation, http://docs.iplanet.com/docs/manuals/directory.html
[13] OpenLDAP 2.0 Administrator's Guide, http://www.openldap.org/doc/admin/