DataGRID

EDG-VOMS-ADMIN DEVELOPER'S GUIDE

DESCRIBING EDG-VOMS-ADMIN RELEASE 0.7

Document identifier:	**edg-voms-admin-dev-guide**
EDMS id:	
Date:	January 14, 2004
Work package:	**WP07: Security**
Partner(s):	**CERN, ELTE**
Lead Partner:	**CERN**
Document status:	**WORKING DRAFT**
Author(s):	Károly Lőrentey, Ákos Frohner
File:	**edg-voms-admin-dev-guide**

Abstract: VOMS is the Virtual Organization Membership Service. This document provides a general overview of the internal architecture of the edg-voms-admin administration service and a description of the database scheme used by that service.

This document is intended to be an introductory overview of edg-voms-admin for developers who need to write client interfaces to the SOAP API, or need to change or debug edg-voms-admin itself. The document is not meant to be a public user guide for the service. Nor it is intended to be a complete, up-to-date guide of the internals of edg-voms-admin; for the exact details, the developer should refer to the API documentation (available in Javadoc), along with the source code of the service itself.

Note that this document describes edg-voms-admin release 0.7; some of the information contained herein may not be applicable to earlier or later releases of the service.

CONTENTS

1. Basic Concepts

In this section, we introduce a number of important concepts that are needed to understand edg-voms-admin.

1.1. Containers

1.2. Fully Qualified Container Names

VOMS defines groups, roles and capabilities. Combinations of the names of these serve as containers for users. The combinations of these names define unique containers.

Let's see some examples of basic containers and their FQCN counterparts:

```
VO           Fred           /Fred
group        production     /Fred/production
group        replicator     /Fred/replicator
role         VO-Admin       /Fred/Role=VO-Admin
role         Admin          /Fred/Role=Admin
capability   long-job       /Fred/Capability=long-job
capability   large-space    /Fred/Capability=large-space
```

In a VOMS credential triplets of these basic containers are returned. Since roles and capabilities can not have subcontainers, we order the groups first in an FQDN.

Let's see a subgroup inside replicator:

```
subgroup     optimisation     /Fred/replicator/optimisation
```

We may add a role name to this, which defines the admins of this subgroup, but not the admins of any other group (or container):

```
/Fred/replicator/optimisation/Role=Admin
```

In summary a FQCN looks like this:

```
/VO[/group[/subgroup(s)]][/Role=role][/Capability=cap]
```

The name has to match the following regexp:

```
^(/[\w-]+)+(/Role=[\w-]+)?(/Capability=[\w\s-]+)?$
```

(`\w` is `[a-zA-Z0-9_]`, `\s` includes the horizontal twhite space characters.)

2. Architectural overview

This section presents a broad overview of the implementation of edg-voms-admin. After reading this section, the reader should have a basic understanding of what goes on inside edg-voms-admin while it processes a request, and she should be able to understand the detailed API documentation embedded in the source as Javadoc comments.

2.1. INTRODUCTION

The implementation of edg-voms-admin follows a basic two-level layered architecture design: SOAP API calls are first translated into high-level *actions* and *questions*, Java objects that represent high-level, user-visible operations on the VO database. The implementation of an action or a question is a problem-domain description of how to execute that operation, including a separate description of the necessary authorization checks. An operation is essentially translated into a series of object manipulations in the problem domain: manipulations of users, groups and other such entities in edg-voms-admin.

The implementation of these database objects comprise the lower layer of edg-voms-admin: it is this layer which translates these low-level object manipulations into concrete SQL statements.

Extreme care was taken to keep the high-level operation descriptions short and clean; there is an elaborate system of helper classes that relieve the burden of transaction management and (most of) error handling from the operation implementations.

2.2. HIGH-LEVEL OPERATIONS: ACTIONS AND QUESTIONS

The edg-voms-admin service defines a number of SOAP APIs which are used by the clients to communicate with the service in a stateless fashion: method calls initiated by a client are executed independently of each other. No state information is retained on the service side between two successive calls of the same client[1]. (Of course, the result of database updates made by earlier calls are visible to later accesses.)

Each SOAP API call is directly mapped to a so-called *operation* by the implementation of the SOAP interface.

Operations are represented by Java objects that define how to perform the operation, and (separately) how to verify that the client is authorized to execute it.

There are two basic kinds of operations. Those operations which change the VO database, but do not need to return a value are called *actions*. `VOMSAdmin.createUser()` and `VOMSAdmin.addMember()` are two examples of SOAP calls that are mapped into actions. In this case, the actions will be instances of the classes `CreateUserAction` and `AddGroupMemberAction` in package `org.edg.security.voms.operation`. Java classes that define action types must implement the `Action` interface, defined in the same package.

Those operations which return a value by querying the database, but do not need to perform changes are called *questions*. Questions must implement the `org.edg.security.voms.operation.Question` interface. The class `org.edg.security.voms.operation.GetUserQuestion` demonstrates a typical question, corresponding to the `VOMSAdmin.getUser()` SOAP API method.

Questions and actions need not care about setting up a database connection, or performing transaction management. All they need to do is call the needed low-level database object manipulations provided by the `org.edg.security.voms.database` package.

For example, the implementation of the `VOMSAdmin.createUser()` SOAP API method looks like this (modulo logging):

```
synchronized public void createUser (User user)
    throws RemoteException {
    Database.perform (new CreateUserAction (user));
}
```

[1]The service-side HTML user interface bypasses the SOAP calls by directly calling the Java methods implementing it, and reuses the security context for the series of edg-voms-admin API calls that are necessary to process a single HTTP request made by the client. This "grouping" of API calls does not affect the behaviour of the service; the core of edg-voms-admin still executes each API call as if they came from separate clients.

(See `org.edg.security.voms.service.admin.VOMSAdminSoapBindingImpl.java`.)

As you can see, all the parameters that are necessary to perform an operation are given to it during instantiation. In this case, the SOAP parameter `user` gets passed on to the action's constructor.

The actual implementation of the create user action is given below:

```
package org.edg.security.voms.operation;
final public class CreateUserAction extends ActionHelper {
    static private final Logger log = Logger.getLogger (CreateUserAction.class);
    /** Parameter: The user to be created. */
    private User user = null;
    public void checkPermission() throws VOMSException {
        DBGroup.getVOGroup().checkPermission (Operation.ADD);
    }
    public void perform() throws VOMSException {
        DBUser u = DBUser.createUser (user.getDN(),
                                      user.getCN(),
                                      DBCA.getInstance (user.getCA()),
                                      user.getCertUri(),
                                      user.getMail());
        log.info ("User " + u.getDN() + " created");
    }
    public CreateUserAction (User user) {
        this.user = user;
    }
```

The implementation of operations should be a self-evident description of what is to be done in the database. We check authorization by requiring that the user is able to do an ADD operation on the VO group. In this case, the actual operation is performed by a simple low-level method call (but note that the CA must be passed as a database object, not just a simple string).

Note the apparent lack of transaction management or even error handling. The operation passes all exceptions that are thrown during its execution to the caller, who is responsible for starting the appropriate transaction before calling the operation's methods, catching any exceptions, and committing the transaction or rolling it back when the action has completed. As doing this right is a tedious procedure that is easy to break, a helper class, `org.edg.security.voms.database.connection.Database` is available to do all this bookkeeping. It is not recommended to bypass the `Database` class by directly calling the `checkPermission()`, `perform()` or `ask()` methods of an operation. This should not be necessary in any circumstances. In fact, one of the reasons for which the concept of operations was added to edg-voms-admin was to make the `Database` class possible.

2.3. Low-Level Database Object Manipulations

As described above, operations essentially translate user-visible high-level SOAP methods into low-level abstract database object manipulations. These objects and their methods for manipulation are implemented by classes in the package `org.edg.security.voms.database`. The names of these classes all have an uppercase `DB` prefix in order to distinguish them from those that are used as parameter and result types of high-level operations.

Instances of these `DB*` classes are abstract representations of one or more rows in a single database table. Their manipulation methods issue low-level SQL statements that query or update the database.

Manipulations that update the database may only be called by Actions. Query manipulations may be called by both Actions and Questions.

A typical update manipulation for creating a new user is shown below. Note that manipulations must thoroughly check their arguments, and must not accept illegal values. (Container names are an exception to this rule; the validity of a container name in the defined naming scheme must be checked by the operation.)

The manipulation retrieves the currently available thread-local database handler from the `CurrentConnection` class, thus the need to explicitly pass database connections to manipulations is eliminated. Another implementation detail to notice is that manipulations are not allowed to throw `SQLException`s. They must convert them to meaningful application-specific errors, or (if all else fails) wrap them up in a `DatabaseError`.

```
/** Creates a new user in the database. */
public static DBUser createUser (String dn, String cn, DBCA ca,
                                 String certUri, String mail)
    throws GeneralDatabaseException, ArgumentException {
    if (dn == null)
        throw new ArgumentException ("User's DN must not be null");
    if (cn == null)
        throw new ArgumentException ("User's CN must not be null");
    if (ca == null)
        throw new ArgumentException ("User's CA must not be null");
    try {
        // Check that user is not in database.
        // TODO: This is clumsy.
        try {
            DBUser old = DBUser.getInstance (dn, ca);
            throw new AlreadyExists (old.toString());
        } catch (NotInDatabase e) {
            // Ignore, this is expected.
        }

        UpdateWrapper u = CurrentConnection.getUpdate();

        // Create the new user.
        PreparedStatement s = u.getStatement
              ("INSERT INTO usr VALUES (?, ?, ?, ?, ?, ?, ?, ?)");
        seq.setToNextval (s, 1);
        s.setString (2, dn);        s.setInt (3, ca.getId ());
        s.setString (4, cn);        s.setString (5, mail);
        s.setString (6, certUri);   s.setLong (7, u.getClientId ());
        s.setLong (8, u.getTransaction ());
        s.executeUpdate ();

        DBUser newUser = null;
        try {
            newUser = DBUser.getInstance (dn, ca);
        }
        catch (NotInDatabase e) {
            throw new InconsistentDatabase
                 ("Just created user \"" + dn + "\" failed to appear in database.");
        }

        // Automatically add her to the VO group.
        // TODO: Do something sensible with vip and cip.
        s = u.getStatement ("INSERT INTO m VALUES (?, ?, NULL, NULL, 0, 0, ?, ?)");
        try {
            s.setLong (1, newUser.getId ());      s.setLong (2, DBGroup.VO_GROUP_ID);
            s.setLong (3, u.getClientId ());      s.setLong (4, u.getTransaction ());
            s.executeUpdate ();
        }
        finally {
            s.close ();
        }

        return newUser;
    }
    catch (SQLException e) {
        throw new DatabaseError (e);
    }
}
```

2.4. TRANSACTION MANAGEMENT

Transaction management in edg-voms-admin is closely related to the handling of database connections. The package `org.edg.security.voms.database.connection` is responsible for database connection management. It contains a simple database connection pool implementation (to be replaced later by standard JNDI connection pools), `ConnectionPool`.

edg-voms-admin defines three types of database connections:

`QueryWrapper` for read-only database manipulations,

`UpdateWrapper` for database update manipulations, and

`DirectUpdate` for direct execution of SQL statements (for special internal purposes).

The first two connection types are used by the `DB*` classes. Update manipulations require an `UpdateWrapper`; queries can work with either an `UpdateWrapper` or a `QueryWrapper`[2].

`DirectUpdate` is for executing SQL statements for special, internal purposes; `DirectUpdate`s are used for e.g. sequence generation (see `org.edg.security.voms.database.Sequence`) or request handling (see `org.edg.security.voms.request.Request`).

Database connections may be allocated from their respective connection pools by static factory methods in the database connection class: `UpdateWrapper.begin()`, `QueryWrapper.get()`, `DirectUpdate.begin()`. Note that these methods are normally not called directly, though: the `Database` helper class takes care of all the work that is needed for accurate transaction management and error handling, including support for transaction restarts.

2.5. REQUEST HANDLING

Requests are entities requesting a certain action in the database. They are submitted by clients who do not have the necessary rights to do the operation themselves. Typically, each new request needs the approval of a VO administrator before the operation is executed. Clients submitting requests may or may not be members of the VO: indeed, requests for VO membership are expected to be the most important request type.

Requests are about the execution of *actions*, in the sense of section 2.2.. Requesting a question is not possible; allowing this would not be particularly useful.

The `Action` instance that the submitter requests to be performed is included in the request itself. Each request has such an instance. If the request is accepted, it executes the action by calling its `perform()`

[2]In fact, queries may work with any of the three connection types, but issuing a query under a `DirectUpdate` is strongly discouraged.

method. An action normally has its own set of parameters that are stored as attributes of the action object, as seen in section 2.2..

Each request has a request type that defines the exact workflow of the request. *Request types* are classes derived from `org.edg.security.voms.request.Request`. The attributes of a concrete request class define the parameters of the request. (Typical request parameters are the id of the client who submitted the request, the action that is requested by the client, etc.) Requests in VOMS are basically state machines – the request type defines the available states and the behaviour of the machine with respect to incoming events.

The *states* of these state machines are instances of special inner classes of the request type class, derived from `org.edg.security.voms.request.Request.State`. The set of possible states of a request type define its behaviour with respect to incoming events. If needed, states can access the request parameters inside the enclosing request instance by means of lexical closure.

Requests maintain a chronicle of things that happened to them. The chronicle is a list of timestamps, client identifications and HTML event descriptions that can be retrieved by calling `Request.getChronicle()`. The chronicle is intended to be presented to the user when she is handling (accepting, denying, etc.) the request.

Requests are stored in the database as serialized Java objects. To speed up request handling, some of the request information is extracted to database-native, indexable columns.

Events are instances of classes derived from the abstract `org.edg.security.voms.request.Event` class. These objects describe the various outside events that may happen to a request (e.g., an administrator accepts the request, or a timeout happens). If a request is given a new event, it forwards it to its current state, which decides what to do with the event. The processing of an event normally involves the transition to a new state.

3. DATABASE SCHEMA

The information that comprises the VO is stored in a MySQL relational database using its transaction-safe InnoDB table backend. The database is queried through a C server (not described here) and modified through a Java service.

In this section, we describe the database schema used by VOMS. After reading this section, the reader should feel comfortable with operating the underlying database of edg-voms-admin. The information contained in this section should be enough to debug and fix small database inconsistencies as reported by edg-voms-admin.

3.1. OVERVIEW

Data tables These tables contain the actual data that define the VO. Changes to these tables are archived for traceability.

Name	*Description*
ACL	Access control lists (ACL) of containers.
GROUPS	List of groups in the VO.
M	Mapping between members and containers (groups, roles and capabilities).
ROLES	List of roles in the VO.
CAPABILITIES	List of capabilities in the VO.
USR	List of users in the VO.

Archive tables These tables contain old data that was deleted or changed in data tables. The definition of archive tables is similar to their corresponding data tables, but extended to support recording of expiration time and administrator information. This archive of changes will be used to implement the edg-voms-admin traceability extensions.

Name	*Description*
ACLD	Archive of container ACLs.
GROUPSD	Archive of groups in the VO.
MD	Archive of the mapping between members and containers.
ROLESD	Archive of roles in the VO.
CAPABILITIESD	Archive of capabilities in the VO.
USRD	Archive of users in the VO.

Service tables These auxiliary tables are used by edg-voms-admin for timekeeping, sequence generation and request handling. Data in these tables is not archived.

Name	*Description*
ADMINS	List of entities referenced in ACLs or createdBy fields.
CA	Known certificate authorities.
REALTIME	Maps transaction ids to timestamps.
SEQUENCES	Holds the values for various monotonically increasing sequences.
REQUESTS	Contains the entire state of the requests submitted to edg-voms-admin.

Unused tables The following tables are not used in the current implementation of edg-voms-admin. Their detailed description is not included in this document. Columns referring to these tables are guaranteed to be set to NULL.

Name	*Description*
VALIDITY	Membership validity.
PERIODICITY	Membership periodicity.
QUERIES	User-defined SQL queries.

3.2. TRACEABILITY FEATURES

Data tables have two columns to identify the admininstrator entity and the transaction number that generated each row in the table.

Column	*Description*
...	...
createdby	The id of the administrator entity that created this row.
createdserial	The serial number of the transaction during which this row was created.

Each data table (except CA) has a corresponding archive table that has the exact same schema as its owner, apart from two extra columns:

Column	*Description*
...	...
createdby	The id of the administrator entity that created this row.
createdserial	The serial number of the transaction during which this row was created.
deletedby	The id of the administrator entity that deleted or updated this row.
deletedserial	The serial number of the transaction during which this row was deleted or updated.

Administrator ids are mapped to DN/CA pairs in the **ADMINS** table. Note that some rows may be created automatically, without user interaction. In this case, **createdby** and **deletedby** point to **ADMINS** entries with a DN that starts with the special prefix `/O=VOMS/O=System` (this prefix is defined in `org.edg.security.voms.service.Constants.INTERNAL_DN_PREFIX`). None of the above traceability columns may be NULL.

3.3. Individual Table Descriptions

In this section, we briefly describe each table in an order that minimizes the number of necessary cross-references.

3.3.1. The sequences table

The **SEQUENCES** table holds the current value of a number of monotically increasing integer sequences that are used throughout the database for generation of unique numerical ids.

Column	*Description*
name	The name of the sequence. This is used in the Java code to refer to the sequence.
value	The last value that was issued by this sequence. The next value will be the next integer after the current value.

```
  name          VARCHAR(32) NOT NULL,
  VALUE         BIGINT,
  INDEX (name)
) TYPE=InnoDB;
```

If the database was modified by hand, it may happen that the database administrator created a new row in one of the data tables but neglected to increment the corresponding sequence value in **SEQUENCES**. edg-voms-admin is robust enough to tolerate and fix such small database inconsistencies by updating sequence values during its startup procedure. It makes not attempt to detect these inconsistencies after the startup has completed.

Perhaps the most important sequence that is defined in this table is `transaction`, which is where the serial number of each update transaction comes from. The `edg-voms-admin-configure` script initializes this sequence as follows:

3.3.2. The realtime table

The **REALTIME** table maps transaction serial numbers to timestamps. It is useful for finding out when a particular transaction occured. As it would be inefficient and unnecessary to store a timestamp for each transaction number, edg-voms-admin only writes a new row to this table if a given amount of time or number of transactions have passed since the last timestamp was written. (The details of this behaviour are user-configurable.)

Column	*Description*
transaction	The serial number of the transaction.
time	The time when the transaction started.

```
  TRANSACTION BIGINT UNSIGNED NOT NULL,
  TIME        TIMESTAMP NOT NULL,
  PRIMARY KEY(TRANSACTION),
  INDEX (TIME)
) TYPE=InnoDB;
```

3.3.3. THE ADMINS TABLE

The **ADMINS** table contains details about each entity that appears as a subject in an access control list, has ever modified the database or handled a request. The **createdby** and **deletedby** columns that are defined in data tables refer to this list of administrators.

There are two kinds of administrator entities that may appear in this table: those corresponding to a certificate issued by a known CA (individual administrators), and those corresponding to groups, roles or capabilities of a VO (collective administrators). (Any VO is allowed, not just the one that is defined by this database. Collective administrators may only be the subject of ACL entries, they may not be referred to in **createdby** or **deletedby** values. Collective administrators are distinguished from individual administrators by special DN/CA values. See the `org.edg.security.voms.service.Constants` class for a list of such special CA values; the DN is generally the fully qualified name of the group/role/etc. that is referred to.

Do not confuse this table with the **USR** table; the contents and semantics of **ADMINS** are independent of the list of users in the VO in any given instant. A VO user normally does not need to administer the VO, and thus does not appear in the **ADMINS** table[3]. Similarly, it is not necessary for an administrator to be a member of the VO in order to modify the database.

Note that having an entry in the **ADMINS** table is not a prerequisite requirement for modifying the database; new **ADMINS** rows are automatically created when needed.

Column	*Description*
adminid	The administrator entity's identifier.
dn	The distinguished name of the administrator entity.
ca	A reference to the CA of the administrator entity.
createdby	The administrator entity that created this entry.
createdserial	The serial number of the transaction during which this entry was created.

```
  adminid BIGINT NOT NULL,
  dn VARCHAR(250) NOT NULL,
  ca SMALLINT UNSIGNED NOT NULL,
  createdby BIGINT UNSIGNED NOT NULL,
  createdserial BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (adminid),
  UNIQUE KEY admin (dn,ca)
) TYPE=InnoDB;
```

The **ADMINS** table contains two entries by default; one is the dummy administrator entry for local database administrators (who do not have to authenticate themselves to modify the database), and one for the default collective administrator named VO-Admin:

```
INSERT INTO admins (adminid, dn, ca, createdBy, createdSerial) VALUES
        (1, "/O=VOMS/O=System/CN=Local␣Database␣Administrator", 1, 1, 0);

-- The VO-Admin role to allow remote administration
INSERT INTO admins (adminid, dn, ca, createdBy, createdSerial) VALUES
        (2, "/$voname/Role=VO-Admin", 4, 1, 0);
```

(Here, `$voname` is replaced by the name of the VO.) The local database administrator is also used for database changes that are done automatically by the edg-voms-admin service.

[3]A normal user may appear in **ADMINS** if she has previously submitted a request.

3.3.4. The CA table

The **CA** table contains a list of certificate authorities that are known to this VO. Each user and administrator of the VO must have a certificate from a CA that appears in this list. Other than that, this table is purely for reference purposes, it has no other role in access control. The set of CA certificates that are accepted for authentication over SSL is determined by a process that is entirely independent of this table. (Although normally this set is a subset of the contents of this table.)

New entries are automatically added to this table when a periodically running background daemon thread finds new certificates in the `/etc/grid-security/certificates/` filesystem directory (another directory may be specified by the `voms.cafiles` configuration property). Note that nothing happens to this table if a CA certificate is removed from that directory: expired CAs remain listed indefinitely.

edg-voms-admin intentionally defines no user-visible operations for manipulating this table, other than the above automatic mechanism for discovering new CAs.

Column	*Description*
cid	The internal id of the CA.
ca	The distinguished name of the CA.
cadescr	A textual description of the CA, intended to be presented to users of the edg-voms-admin client interfaces.

```
  cid SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  ca VARCHAR(250),
  cadescr VARCHAR(250),
  PRIMARY KEY  (cid),
  UNIQUE KEY ca (ca)
) TYPE=InnoDB;
```

The **CA** table also contains entries for several "virtual CAs" that have no corresponding CA certificate. These entries are created by the `edg-voms-admin-configure` script during database installation and are used in the CA fields of collective administrator entities, and the administrator entity of the *local database administrator* who accesses the database without authentication.

```
        1, "/O=VOMS/O=System/CN=Dummy␣Certificate␣Authority",
        "A␣dummy␣CA␣for␣local␣database␣maintenance.");
INSERT INTO ca (cid, ca, cadescr) VALUES (
        2, "/O=VOMS/O=System/CN=Authorization␣Manager␣Attributes",
        "A␣virtual␣CA␣corresponding␣to␣authz␣manager␣attributes");
INSERT INTO ca (cid, ca, cadescr) VALUES (
        3, "/O=VOMS/O=System/CN=VOMS␣Group",
        "A␣virtual␣CA␣corresponding␣to␣a␣VO␣group");
INSERT INTO ca (cid, ca, cadescr) VALUES (
        4, "/O=VOMS/O=System/CN=VOMS␣Role",
        "A␣virtual␣CA␣corresponding␣to␣a␣VO␣role");
INSERT INTO ca (cid, ca, cadescr) VALUES (
        5, "/O=VOMS/O=System/CN=VOMS␣Capability",
        "A␣the␣virtual␣CA␣corresponding␣to␣a␣VO␣capability");
```

3.3.5. The USR table

The **USR** table contains the list of users in the VO. Users are never created automatically; the database is created with an empty **USR** table.

Column	*Description*
uid	The internal id of the user.
dn	The distinguished name of the user, as it appears in her certificate.
ca	The CA that issued the user's certificate.
cn	The common name of the user.
mail	The email address of the user. Optional.
cauri	An optional pointer to the user's certificate.
createdby	The administrator entity that created this entry.
createdserial	The serial number of the transaction during which this entry was created.

```
  uid BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
  dn VARCHAR(250) NOT NULL,
  ca SMALLINT UNSIGNED NOT NULL,
  cn VARCHAR(250) NOT NULL,
  mail VARCHAR(250) DEFAULT NULL,
  cauri VARCHAR(250) DEFAULT NULL,
  createdby BIGINT UNSIGNED NOT NULL,
  createdserial BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY  (uid),
  UNIQUE KEY dnca (dn,ca),
  KEY dn (dn)
) TYPE=InnoDB;
```

3.3.6. THE GROUPS TABLE

The **GROUPS** table contains the list of groups in the VO, starting with the all-encompassing VO group, which is created during database initialization.

```
INSERT INTO groups (gid, dn, parent, aclid, defaultAclid, must,
        createdBy, createdSerial) VALUES (1, "/$voname", 1, 1, 2, 1, 1, 0);
```

(Here, `$voname` is replaced by the name of the VO.) The VO group is a system group, it may not be deleted.

Column	*Description*
gid	The internal id of the group.
dn	The fully qualified group name.
parent	The id of the parent of the group. The parent of the VO group is itself.
aclid	The id of the group's access control list.
defaultid	The id of the access control list that is used to initialize the ACL's of the sub-groups of this group.
must	A non-null value means the group is not deniable. The current edg-voms-admin implementation always sets this value to 1.
createdby	The administrator entity that created this entry.
createdserial	The serial number of the transaction during which this entry was created.

```
  gid BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
  dn VARCHAR(255) NOT NULL,
  parent BIGINT UNSIGNED NOT NULL,
  aclid BIGINT UNSIGNED NOT NULL,
  defaultaclid BIGINT UNSIGNED NOT NULL,
  must tinyint DEFAULT NULL,
  createdby BIGINT UNSIGNED NOT NULL,
  createdserial BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (gid),
  KEY parentg (parent),
  KEY groupname (dn)
) TYPE=InnoDB;
```

3.3.7. THE ROLES TABLE

The **ROLES** table contains the list of roles in the VO. The script `edg-voms-admin-configure` creates a role for remote VO administration during database initialization:

```
INSERT INTO roles (rid, role, aclid, createdBy, createdSerial) VALUES
        (1, 'VO-Admin', 3, 1, 0);
```

(Here, `$voname` is replaced by the name of the VO.) This role is not a system group and is not handled differently from any other role; the administrators are allowed to freely delete this role.

Column	*Description*
rid	The internal id of the role.
role	The role name, qualified by the VO group.
aclid	The id of the access control list that applies to the role.
createdby	The administrator entity that created this entry.
createdserial	The serial number of the transaction during which this entry was created.

```
  rid BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
  role VARCHAR(255) NOT NULL,
  aclid BIGINT NOT NULL,
  createdby BIGINT UNSIGNED NOT NULL,
  createdserial BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY  (rid),
  KEY role (role)
) TYPE=InnoDB;
```

3.3.8. THE CAPABILITIES TABLE

This table holds the list of capabilities that are known to the VO.

Note that the support for capabilities in edg-voms-admin is still experimental.

Column	*Description*
cid	The internal id of the capability.
capability	The name of the capability.
aclid	The id of the access control list that applies to the capability.
createdby	The administrator entity that created this entry.
createdserial	The serial number of the transaction during which this entry was created.

```
  cid BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
  capability VARCHAR(255) NOT NULL,
  aclid BIGINT NOT NULL,
  createdby BIGINT UNSIGNED NOT NULL,
  createdserial BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY  (cid),
  KEY capability (capability)
) TYPE=InnoDB;
```

3.3.9. THE M TABLE

The **M** table defines membership relations in the VO. Each VO user is a member of the VO group, and may be a member of any number of other groups. A user may have any number of roles, each of which is qualified with the group where the role applies. Possible combinations (X denotes that the given field is present, - denotes that the field is NULL):

Group	*Role*	*Cap.*	Description
-	any	any	Invalid. (Not generated by edg-voms-admin.)
X	-	-	The user is a member of the given group.
X	X	-	The user is a member of the given group, and also has the specified role in that group.
X	-	X	The user is a member of the given group, and also has the given capability in that group.
X	X	X	Invalid. (Not generated by edg-voms-admin.)

Capabilities are only bound to the VO group. Note that the support for capabilities in edg-voms-admin is still experimental.

Column	*Description*
uid	The id of a user.
gid	The id of a group.
rid	The id of a role. (Optional.)
cid	The id of a capability. (Optional.)
vid	The id of a validity. (Unused, always zero.)
pid	The id of a periodicity. (Unused, always zero.)
createdby	The administrator entity that created this entry.
createdserial	The serial number of the transaction during which this entry was created.

```
  uid BIGINT UNSIGNED NOT NULL references usr(uid),
  gid BIGINT UNSIGNED NOT NULL references groups(gid),
  rid BIGINT UNSIGNED references roles(rid),
  cid BIGINT UNSIGNED references capabilities(cid),
  vid BIGINT UNSIGNED references validity(vid),
  pid BIGINT UNSIGNED references periodicity(pid),
  createdby BIGINT UNSIGNED NOT NULL,
  createdserial BIGINT UNSIGNED NOT NULL,
  UNIQUE m (uid,gid,rid,cid),
  KEY uid (uid),
  KEY rid (rid),
  KEY cid (cid),
  KEY container (gid,rid,cid)
) TYPE=InnoDB;
```

3.3.10. THE ACL TABLE

The **ACL** table contains the entries of the ACLs of the various containers in the VO. It is not an error for an ACL to be empty. Such empty ACLs have no entries, and thus have not a single row in this table.

Column	*Description*
aid	The internal identifier of the ACL.
adminid	The id of the administrator entity that is the subject of this ACL entry.
operation	The id of the operation that this entry allows or denies.
allow	A zero value means the operation is denied; a one means the operation is allowed.

```
  aid BIGINT UNSIGNED NOT NULL,
  adminid BIGINT NOT NULL,
  operation SMALLINT NOT NULL,
  allow tinyint NOT NULL, -- 0 means deny, 1 means allow
  createdby BIGINT UNSIGNED NOT NULL,
  createdserial BIGINT UNSIGNED NOT NULL,
  INDEX (aid),  -- not primary key!
  INDEX (aid, adminid, operation)
) TYPE=InnoDB;
```

3.3.11. The requests table

The **REQUESTS** table contains all the requests that are currently processed by edg-voms-admin. The requests are stored in their serialized Java object form. To allow efficient access to requests, some important request parameters are extracted to separate, indexed columns.

Note that the support for requests is still experimental, and the details are to change.

Column	*Description*
reqid	The id of the request.
complete	True if the request is in a final state, and thus no more state changes are expected to occur.
state	The name of the current state of the request.
client	The id of the administrator entity that submitted this request.
lastchange	The time when the request last processed an event.
request	The complete request, in a serialized Java object form.

```
  reqid          BIGINT NOT NULL,
  complete       BOOL,
  state          VARCHAR(32),
  client         BIGINT,
  lastchange     TIMESTAMP,
  request        BLOB NOT NULL,
  PRIMARY KEY(reqid),
  INDEX (reqid)
) TYPE=InnoDB;
```

4. Glossary

Attribute: a string which is returned by VOMS for a user. An attribute can be a variety of things (e.g. a group, a role etc.), but they only differ in the behaviour, how they are assigned to the user.

Container: a set of users (see also: Group, Role, Capability). (It is like a directory in a filesystem.)

Group: a group is a container which is unconditionally assigned to its member users. A group may contain other groups, roles and users.

Groups can be organised in a tree like structure: no cycles or shared subgroups are allowed. There is one group at the top of this hierarchy, which represents the VO itself. It is called the VO group.

If a new group is created inside an existing one, then it will inherit its parent name:

```
parent: /CERN/Atlas
groupname: designers
FQGN: /CERN/Atlas/designers
```

The group's unique name is this *fully qualified group name*.

The ACL of newly created groups and roles is a copy of the default ACL of its parent.

A member of a group has to be a member of its parent group as well.

VO group: a group which represents the VO.

This is the root of the group tree, which implies that every user in the database must be also a member of this group.

DB representation: The VO group has a **gid** value of 1.

Role: a role is a container which is only returned as an attribute if a member user explitly asks for it.

Membership in a role is always associated with a group: a user may have role R in group A, but not in group B.

Roles may have no sub-roles. Each role name must be unique.

Capability: a capability is a free-form string which can be assigned to any user. It is always returned in the VOMS credential.

The names of the capabilities must be unique. Like in the case of roles, the namespace of the capabilities is flat; there is no support for sub-capabilities.

User: a user is an entity who can be member of the various containers. A user must be registered in the VO before it can be assigned to any container.

CA: a certificate authority is an entity which issues user credentials. A CA guarantees the uniqueness of the user names only inside its own domain, thus we have to add this property to every user to be able to uniquely identify them.

Each user must have a valid associated certificate authority, which grants its identity.

The CA table is a view on the `/etc/grid-security/credentials` (default place) directory, so whenever a new certificate turns up there, it is added to the table. This update procedure runs periodically during the operation of the service. Once a CA is added, it will stay in the CA table indefinitely.

ACL: an access control list is a set of access control entries which is assigned to a group, role or capability. The entries of an ACL are in OR relation.

ACL entry: an ACL entry is a tuple which contains an operation, a principal and an allow/deny flag. A principal can be any user, group or role in the database. An operation can be:

- create/delete – controls subgroup operations
- add/remove – controls membership operations
- setACL/getACL – controls ACL operations
- setDefault/getDefault – controls default membership operations
- ALL – special permission for all operations

(see the details at the operations)

The syntax used in this document is: $(+/-) : \langle principal \rangle : \langle operation \rangle$

For example if Joe is allowed to add a new member to a group: $+ : Joe : add$

Authorization: a check made to determine if a previously authenticated remote client has the rights necessary to execute a given operation.

The process first looks up the access control list that governs the requested operation (this likely depends on the object of the operation). Then, edg-voms-admin gathers the available attributes of the client:

- authentication information: id from the client's certificate
- any attributes from the certificate's VOMS extension
- if the client is a member of the local VO, then edg-voms-admin automatically looks up her attributes in the local database

After these preliminary steps, edg-voms-admin compares the ACL entries with the gathered attributes. If there is a matching allow entry and there is no matching deny entry, then the authorization is successful:

```
granted = no
for every user-attr in list-of-user-attributes
    for every entry in access-control-list
        if (entry.principal = user-attr and
              entry.operation = operation) then
            if entry.allow then
                granted = yes
            else
                return no
            end-if
        end-if
    end-for
end-for
return granted
```