XUUDB MANUAL

UNICORE Team

Document Version:	2.1.1
Component Version:	2.6.0
Date:	23 02 2021

This work is co-funded by the EC EMI project under the FP7 Collaborative Projects Grant Agreement Nr. INFSO-RI-261611.

Contents

The XUUDB server is an Attribute Source implementation which can be used by UNICORE servers. It is used to map a UNICORE identity (an X500 distinguished name) to authorisation and incarnation attributes. The XUUDB is also capable of performing dynamic mappings of incarnation attributes, using a rule engine.

For more information about UNICORE visit https://www.unicore.eu.

1 Overview

The UNICORE XUUDB is used to map a UNICORE user identity (an X.500 distinguished name (DN)) to a set of attributes. The attributes are typically used to provide local account details (uid, gid(s)) and sometimes also to provide authorization information: user's role.

The UNICORE XUUDB is best suited as a site-service. Theoretically it can be used for multiple sites, however as it offers limited authorization capabilities and doesn't allow for grouping users, it is better to use the more flexible Unity server in such a case. In case of the simple one host-service XUUDB sometimes can be replaced by a simple file storing the mappings. Please refer to the UNICORE/X documentation for more information.

The XUUDB offers two web services, one for querying, and one for administration of the users' database. There are several clients which can use the XUUDB server:

- Admin client (see Section 4) can be used to control the XUUDB database contents.
- UNICORE servers include the XUUDB client code (it is named XUUDB Attribute Information Point) and can consume and process the XUUDB information.

Both admin and client access to the XUUDB can be protected by a client-authenticated SSL.

The XUUDB can map users using two different mechanisms:

- *classic* or *static* mechanism, where administrator enters mappings for each DN manually,
- *dynamic* mechanism, where administrator only define rules stating what attributes should be assigned to UNICORE users fulfilling rule's condition.

1.1 The classic mapping

The classic or static mechanism when UNICORE is used as a gateway to HPC site, with a well defined set of users. It is also useful in federated scenarios when a dedicated, external infrastructure is build to maintain a global list of users.

Using it it is possible to set a list of UNIX logins (aka XLogins or uids), a list of UNIX groups (aka projects or gids) and the role attribute used for authorization. The first uid and the first gid is assumed to be the default one but Grid users are allowed to choose any of the available.

In case of the default authorization policy the *user* role is required to get a normal access to the site, the *admin* role grants super-user privileges, and the *banned* role bans the user.

The XUUDB stores and compares only distinguished names (DNs), not full certificates.

Multiple UNICORE sites can share the XUUDB, even if the attributes are different per UNICORE site. Sites are grouped by the so-called GCID (grid component ID).

1.2 The dynamic mapping

The dynamic mechanism is used to map users who *were already authorized*, therefore it doesn't make sense (and is not possible) to assign the authorization attributes as *role*. The dynamic mechanism is useful in deployments where a site doesn't know the precise list of its users (which are maintained externally), or simply doesn't want to define local accounts for each UNICORE user. In other words, the site relies on a trusted 3rd party to maintain a list of genuine and authorized users, and automatically assigns a local account to each user.

As it will be shown later on dynamic mappings can be also used in other scenarios, also being complementary to static mappings.

Dynamic mappings configuration is described in the section Section 3.4.

IMPORTANT NOTE ON PATHS

XUUDB is distributed either as an platform independent and portable bundle (as part of the UNICORE quickstart package) or as an installable, platform dependent package such as RPM.

Depending on the installation package used paths are different. If installing using distribution-specific package the following path prefixes are used:

```
CONF=/etc/unicore/xuudb
BIN=/usr/sbin
ADMIN=/usr/sbin/unicore-xuudb-admin
LOG=/var/log/unicore/xuudb
```

If installing using portable bundle all XUUDB files are installed under a single directory. Path prefixes used then are as follows, where INST is a directory where the XUUDB was installed:

```
CONF=INST/conf
BIN=INST/bin
ADMIN=BIN/admin.sh
LOG=INST/log
```

The above variables (CONF, BIN, ADMIN and LOG) are used throughout the rest of this manual.

---

2 Installation

The UNICORE XUUDB is distributed in the following formats:

1. As a part of a platform independent installation bundle called UNICORE core server bundle.
2. As a platform-specific binary package available for RedHat (Centos) and Debian platforms. Those packages are not tested on all possible platforms, but should work without any problems with other versions of similar distributions.

In both cases an installation of XUUDB installs both XUUDB Server and XUUDB admin client.

After installing the server you will have to configure it. This is described in the section Section 3.

2.1 Installation from Core Server Bundle

Download the core server bundle from the UNICORE project website at https://sourceforge.net/-projects/unicore/files/Servers/Core

2.2 Installation from RPM or DEB packages

An up-to-date list of the available repositories is given on the following page https://sourceforge.net/-p/unicore/wiki/Linux_Repositories/

The preferred way is to use Yum or APT to install (and subsequently update) XUUDB.

After the repository is configured, the following command installs XUUDB:

```
$> yum install unicore-xuudb
```

or in case of Debian systems

```
$> apt-get install unicore-xuudb
```

3 The XUUDB server

3.1 Security

Usually, client-authenticated SSL is used to protect the XUUDB. For this you will need certificates for the XUUDB server and all Grid components that want to talk to the XUUDB. In general the UNICORE servers (like UNICORE/X) and the XUUDB-admin client need to connect to the XUUDB-server. To make SSL connections possible, you have to put the following certificates as trusted certs into the XUUDB's server truststore:

- CA certificate(s) of the UNICORE/X server(s) that query the XUUDB
- CA certificate(s) of the XUUDB-admin user certificate(s)

and XUUDB's CA certificate in the truststores of its clients.

XUUDB server may be run using a plain HTTP port. Then there is no access control at all, so this mode is useful only in environments where XUUDB port is fully protected otherwise against unauthorised access.

3.2 Administrative access

The XUUDB provides two kinds of web service interfaces, one for querying the XUUDB (i.e. mapping UNICORE users to local UNIX users), and a second one for administration of the XUUDB (i.e. adding and editing entries). All access to the XUUDB (including the administration utility!) is through these web services. To prevent arbitrary users from modifying the XUUDB, the administrative interface has to be protected.

To protect the administrative interface, an ACL file is used, which is a plain text file containing the distinguished names of the administrators. At least, it has to contain the DN of the certificate used by the administration utility.

As the static XUUDB data is rather sensitive (at least if privacy of the users is a concern) and dynamic mappings often require some local modifications (e.g. assigning an account from a pool) it is often desirable to protect also the query operations. The XUUDB server offers such an option (see Section 3.3.1).

The ACL file can be changed at runtime to easily add or remove administrators.

To change the location of the ACL file, edit the server configuration and set a configuration parameter (see Section 3.3.1).

The ACL entries are expected in the RFC 2253 format. To get the name of a certificate in the correct format using openssl, you can use the following OpenSSL command:

```
$> openssl x509 -in demouser.pem -noout -subject -nameopt  ↩
   RFC2253
```

3.3 Configuration

By default, the configuration is defined in the file `CONF/xuudb_server.conf`. To use a different configuration file, edit the start script, or use `--start <config_file>` as command line arguments when starting.

The server's configuration file allows for setting the general XUUDB settings, database backend settings, advanced HTTP server settings and finally (for secure HTTPS URLs) the server's truststore and credential. The available properties are described in the following sections.

For production deployments you should review the listen address and setup correctly truststore and credential. Defaults for the embedded database configuration and HTTP server settings are usually fine. In case if you plan to use dynamic mappings, also the dynamic mapping rules need to be provided.

3.3.1 Base server settings

Property name	Type	Default value / mandatory	Description
`xuudb.aclFile`	filesystem path	-	File with DNs of clients authorised to access protected XUUDB services.
`xuudb.address`	string	`http://localhost:8443`	HTTPS or HTTP URL where the server should listen.
`xuudb.db.[.*]`	string *can have subkeys*	-	Properties with this prefix are used to configure database backend, used by XUUDB. See separate documentation for details.
`xuudb.dynamicAttributesConfig`	filesystem path	`conf/dynamicAttributesConfig.xml`	File with configuration of the dynamic part of the XUUDB.
`xuudb.protectAll`	[true, false]	`false`	If true then access to both query and modify operations are protected by ACL. If false then only modification operations are protected.
`xuudb.type`	[dn]	`dn`	DEPRECATED and no longer required.

3.3.2 Database settings

The XUUDB can be configured to use different database backends. Currently an embedded H2 database and external MySQL are supported. H2 database (the default) requires no additional configuration actions. In any case XUUDB will automatically create the required database tables.

For MySQL you have to properly set up the server and create a database. After installing and starting the MySQL server login to its using MySQL client as administrator and using a commands similar to the below ones, create a database and assign full access to a xuudb user.

```
create database xuudb;
```

```
grant all on xuudb.* to 'xuudbuser'@'127.0.0.1' identified ↩
   by 'pass';
```

Of course you are free to choose different names for the user, password and database. If XUUDB server is installed on other host then the proper address must be set instead of localhost.

Use the following properties to configure database connection from the XUUDB server. In case of external database pay attention to enter proper values.

Property name	Type	Default value / mandatory	Description
		--- Database ---	
`xuudb.db.dialect`	[h2, mysql]	`h2`	Database SQL dialect. Must match the selected driver, however sometimes more then one driver can be available for a dialect.
`xuudb.db.driver`	Class extending java.sql.Driver	`org.h2.Driver`	Database driver class name. This property is optional - if not set, then a default driver for the chosen database type is used.
`xuudb.db.jdbcUrl`	string	`jdbc:h2:data/xuudb`	Database JDBC URL.
`xuudb.db.password`	string	*empty string*	Database password.
`xuudb.db.username`	string	`sa`	Database username.

3.3.3 Configuring advanced HTTP server settings

UNICORE servers are using an embedded Jetty HTTP server. In most cases the default configuration should be perfectly fine. However, for some sites (e.g. experiencing an extremely high load) HTTP server settings can be fine-tuned with the following parameters.

Property name	Type	Default value / mandatory	Description
`xuudb.httpServer.CORSallowedHeaders`	string	`-`	CORS: comma separated list of allowed HTTP headers (default: any)
`xuudb.httpServer.CORSallowedMethods`	string	`GET,PUT,POST,DELETE,HEAD`	CORS: comma separated list of allowed HTTP verbs.
`xuudb.httpServer.CORSallowedOrigins`	string	`-`	CORS: allowed script origins.

Property name	Type	Default value / mandatory	Description
`xuudb.httpServer.CORS.allowPreflight`	[true, false]	`false`	CORS: whether preflight OPTION requests are chained (passed on) to the resource or handled via the CORS filter.
`xuudb.httpServer.CORS.exposedHeaders`	string	`Location,Content-Type`	CORS: comma separated list of HTTP headers that are allowed to be exposed to the client.
`xuudb.httpServer.disabledCipherSuites`	string	*Empty string*	Space separated list of SSL cipher suites to be disabled. Names of the ciphers must adhere to the standard Java cipher names, available here: http://docs.oracle.com/-javase/8/docs/technotes/-guides/security/-SunProviders.html#SupportedCipherSuites
`xuudb.httpServer.enableCORS`	[true, false]	`false`	Control whether Cross-Origin Resource Sharing is enabled. Enable to allow e.g. accesing REST services from client-side JavaScript.
`xuudb.httpServer.enableHSTS`	[true, false]	`false`	Control whether HTTP strict transport security is enabled. It is a good and strongly suggested security mechanism for all production sites. At the same time it can not be used with self-signed or not issued by a generally trusted CA server certificates, as with HSTS a user can't opt in to enter such site.
`xuudb.httpServer.fastRandom`	[true, false]	`false`	Use insecure, but fast pseudo random generator to generate session ids instead of secure generator for SSL sockets.

Property name	Type	Default value / mandatory	Description
xuudb.httpServer.gzip.enable	[true, false]	false	Controls whether to enable compression of HTTP responses.
xuudb.httpServer.gzip.minGzipSize	integer number	200000	Specifies the minimal size of message that should be compressed.
xuudb.httpServer.highLoadNumber	integer number	200	deprecated
xuudb.httpServer.lowResourceMaxIdleTime	integer >= 1	1000	deprecated
xuudb.httpServer.maxConnections	integer >= 0	0	Maximum number of incoming connections to this server. If set to a value larger than 0, incoming connections will be limited to that number. Default is 0 = unlimited.
xuudb.httpServer.maxIdleTime	integer >= 1	200000	Time (in ms.) before an idle connection will time out. It should be large enough not to expire connections with slow clients, values below 30s are getting quite risky.
xuudb.httpServer.maxThreads	integer number	255	Maximum number of threads to have in the thread pool for processing HTTP connections. Note that this number will be increased with few additional threads to handle connectors.
xuudb.httpServer.minThreads	integer >= 1	1	Minimum number of threads to have in the thread pool for processing HTTP connections. Note that this number will be increased with few additional threads to handle connectors.
xuudb.httpServer.requireClientAuth	[true, false]	true	Controls whether the SSL socket requires client-side authentication.
xuudb.httpServer.wantClientAuth	[true, false]	true	Controls whether the SSL socket accepts (but does not require) client-side authentication.

Property name	Type	Default value / mandatory	Description
`xuudb.httpServer.xFrameAllowed`	String	`http://localhost`	URI origin that is allowed to embed web interface inside a (i)frame. Meaningful only if the xFrameOptions is set to *allowFrom*. The value should be in the form: *http[s]://host[:port]*
`xuudb.httpServer.xFrameOptions`	[deny, sameOrigin, allowFrom, allow]	`deny`	Defines whether a clickjacking prevention should be turned on, by insertionof the X-Frame-Options HTTP header. The *allow* value disables the feature. See the RFC 7034 for details. Note that for the *allowFrom* you should define also the xFrameAllowed option and it is not fully supported by all the browsers.

Example

> **Note**
> Various UNICORE modules use different property prefixes. Here we don't put any, but in practice you have to use the prefix (see the reference table above for the actual prefix). Also properties might need to be provided using different syntax, as XML.

In this example we will turn on compression of all responses bigger then 50kB (assuming that the client supports decompression). Additionally we are limiting the number of concurrent clients that can be served to more or less 50, while keeping 10 threads ready all the time to server new clients quicker.

```
jetty.gzip.enable=true
jetty.gzip.minGzipSize=51200
jetty.maxThreads=50
jetty.minThreads=10
```

3.3.4 Configuring PKI trust settings

Public Key Infrastructure (PKI) trust settings are used to validate certificates. This is performed, in the first place when a connection with a remote peer is initiated over the network, using the SSL (or TLS) protocol. Additionally certificate validation can happen in few other situations, e.g. when checking digital signatures of various sensitive pieces of data.

Certificates validation is primarily configured using a set of initially trusted certificates of so called Certificate Authorities (CAs). Those trusted certificates are also known as *trust anchors* and their collection is called a *trust store*.

Except of *trust anchors* validation mechanism can use additional input for checking if a certificate being checked was not revoked and if its subject is in a permitted namespace.

UNICORE allows for different types of trust stores. All of them are configured using a set of properties.

- *Keystore trust store* - the only format supported in older UNICORE versions. Trusted certificates are stored in a single binary file in JKS or PKCS12 format. The file can be only manipulated using a special tool like JDK *keytool* or openssl (in case of PKCS12 format). This format is great if trust store should be in a single file or when compatibility with other Java solutions or older UNICORE releases is desired.
- *OpenSSL trust store* - allows to use a directory with CA certificates stored in PEM format, under precisely defined names: the CA certificates, CRLs, signing policy files and namespaces files are named <hash>.0, <hash>.r0, <hash>.signing_policy and <hash>.namespaces. Hash is the old hash of the trusted CA certificate subject name (in Openssl version > 1.0.0 use -suject_hash_old switch to generate it). If multiple certificates have the same hash then the default zero number must be increased. This format is the same as used by other then UNICORE popular middlewares as Globus and gLite. It is suggested when a common trust store with such middlewares is needed.
- *Directory trust store* - the most flexible and convenient option, suggested for all remaining cases. It allows to use a list of wildcard expressions, concrete paths of files or even URLs to remote files as a set of trusted CAs and in the same way for the CRLs. With this trust store administrator can simply configure all files (or all with a specified extension) in a directory to be used as a trusted certificates.

In all cases trust stores can be (and by default are) configured to be automatically refreshed.

Property name	**Type**	**Default value / mandatory**	**Description**
`xuudb.truststore.allowProxy`	[ALLOW, DENY]	`ALLOW`	Controls whether proxy certificates are supported.
`xuudb.truststore.type`	[keystore, openssl, directory]	*mandatory to be set*	The truststore type.

Property name	Type	Default value / mandatory	Description
xuudb.truststore.updateInterval	integer number	600	How often the truststore should be reloaded, in seconds. Set to negative value to disable refreshing at runtime. *(runtime updateable)*
--- Directory type settings ---			
xuudb.truststore.directory.connectionTimeout	integer number	15	Connection timeout for fetching the remote CA certificates in seconds.
xuudb.truststore.directory.diskCachePath	filesystem path	-	Directory where CA certificates should be cached, after downloading them from a remote source. Can be left undefined if no disk cache should be used. Note that directory should be secured, i.e. normal users should not be allowed to write to it.
xuudb.truststore.directory.encoding	[PEM, DER]	PEM	For directory truststore controls whether certificates are encoded in PEM or DER. Note that the PEM file can contain arbitrary number of concatenated, PEM-encoded certificates.
xuudb.truststore.directoryLocations.*	list of properties with a common prefix	-	List of CA certificates locations. Can contain URLs, local files and wildcard expressions. *(runtime updateable)*
--- Keystore type settings ---			
xuudb.truststore.keystoreFormat	string	-	The keystore type (jks, pkcs12) in case of truststore of keystore type.
xuudb.truststore.keystorePassword	string	-	The password of the keystore type truststore.
xuudb.truststore.keystorePath	string	-	The keystore path in case of truststore of keystore type.
--- Openssl type settings ---			

Property name	Type	Default value / mandatory	Description
xuudb.truststore.openssl.newStoreFormat	[true, false]	false	In case of openssl truststore, specifies whether the trust store is in openssl 1.0.0+ format (true) or older openssl 0.x format (false)
xuudb.truststore.openssl.nsMode	[GLOBUS_EUGRIDPMA, EUGRIDPMA_GLOBUS, GLOBUS, EUGRIDPMA, GLOBUS_EUGRIDPMA_REQUIRE, EUGRIDPMA_GLOBUS_REQUIRE, GLOBUS_REQUIRE, EUGRIDPMA_REQUIRE, EUGRIDPMA_AND_GLOBUS, EUGRIDPMA_AND_GLOBUS_REQUIRE, IGNORE]	GLOBUS_EUGRIDPMA	In case of openssl truststore, controls which (and in which order) namespace checking rules should be applied. The *REQUIRE* settings will cause that all configured namespace definitions files must be present for each trusted CA certificate (otherwise checking will fail). The *AND* settings will cause to check both existing namespace files. Otherwise the first found is checked (in the order defined by the property).
xuudb.truststore.openssl.path	filesystem path	/etc/grid-security/certificates	Directory to be used for opeenssl truststore.
--- Revocation settings ---			
xuudb.truststore.crl.connectionTimeout	integer number	15	Connection timeout for fetching the remote CRLs in seconds (not used for Openssl truststores).
xuudb.truststore.crl.fileSystemCachePath	filesystem path	-	Directory where CRLs should be cached, after downloading them from remote source. Can be left undefined if no disk cache should be used. Note that directory should be secured, i.e. normal users should not be allowed to write to it. Not used for Openssl truststores.

Property name	Type	Default value / mandatory	Description
xuudb.truststore.crl.locations.*	list of properties with a common prefix	-	List of CRLs locations. Can contain URLs, local files and wildcard expressions. Not used for Openssl truststores. *(runtime updateable)*
xuudb.truststore.crl.mode	[REQUIRE, IF_VALID, IGNORE]	IF_VALID	General CRL handling mode. The IF_VALID setting turns on CRL checking only in case the CRL is present.
xuudb.truststore.crl.refresh.interval	integer number	600	How often CRLs should be updated, in seconds. Set to negative value to disable refreshing at runtime. *(runtime updateable)*
xuudb.truststore.ocsp.cache.ttl	integer number	3600	For how long the OCSP responses should be locally cached in seconds (this is a maximum value, responses won't be cached after expiration)
xuudb.truststore.ocsp.cache.path	file system path	-	If this property is defined then OCSP responses will be cached on disk in the defined folder.
xuudb.truststore.ocsp.localResponders.<NUMBER>.*	list of properties with a common prefix	-	Optional list of local OCSP responders
xuudb.truststore.ocsp.mode	[REQUIRE, IF_AVAILABLE, IGNORE]	IF_AVAILABLE	General OCSP ckecking mode. REQUIRE should not be used unless it is guaranteed that for all certificates an OCSP responder is defined.
xuudb.truststore.ocsp.timeout	integer number	10000	Timeout for OCSP connections in miliseconds.
xuudb.truststore.revocation.order	[CRL_OCSP, OCSP_CRL]	OCSP_CRL	Controls overal revocation sources order

Property name	Type	Default value / mandatory	Description
xuudb.truststore.revocationUseAll	[true, false]	false	Controls whether all defined revocation sources should be always checked, even if the first one already confirmed that a checked certificate is not revoked.

Examples

Note
Various UNICORE modules use different property prefixes. Here we don't put any, but in practice you have to use the prefix (see the reference table above for the actual prefix). Also properties might need to be provided using different syntax, as XML.

Directory trust store, with a minimal set of options:

```
truststore.type=directory
truststore.directoryLocations.1=/trust/dir/*.pem
truststore.crlLocations=/trust/dir/*.crl
```

Directory trust store, with a complete set of options:

```
truststore.type=directory
truststore.allowProxy=DENY
truststore.updateInterval=1234
truststore.directoryLocations.1=/trust/dir/*.pem
truststore.directoryLocations.2=http://caserver/ca.pem
truststore.directoryEncoding=PEM
truststore.directoryConnectionTimeout=100
truststore.directoryDiskCachePath=/tmp
truststore.crlLocations.1=/trust/dir/*.crl
truststore.crlLocations.2=http://caserver/crl.pem
truststore.crlUpdateInterval=400
truststore.crlMode=REQUIRE
truststore.crlConnectionTimeout=200
truststore.crlDiskCachePath=/tmp
```

Openssl trust store:

```
truststore.type=openssl
truststore.opensslPath=/truststores/openssl
truststore.opensslNsMode=EUGRIDPMA_GLOBUS_REQUIRE
```

```
truststore.allowProxy=ALLOW
truststore.updateInterval=1234
truststore.crlMode=IF_VALID
```

Java keystore used as a trust store:

```
truststore.type=keystore
truststore.keystorePath=src/test/resources/certs/truststore. ↩
    jks
truststore.keystoreFormat=JKS
truststore.keystorePassword=xxxxxx
```

3.3.5 Configuring the credential

UNICORE uses private key and a corresponding certificate (called together as a *credential*) to identify users and servers. Credentials might be provided in several formats:

- Credential can be obtained from a *keystore file*, encoded in JKS or PKCS12 format.
- Credential can be loaded as a pair of PEM files (one with private key and another with certificate),
- or from a pair of DER files,
- or even from a single file, with PEM-encoded certificates and private key (in any order).

The following table list all parameters which allows for configuring the credential. Note that nearly all options are optional. If not defined, the format is tried to be guessed. However some credential formats require additional settings. For instance if using *der* format the *keyPath* is mandatory as you need two DER files: one with certificate and one with the key (and the latter can not be guessed).

Property name	**Type**	**Default value / mandatory**	**Description**
`xuudb.credential.path`	file system path	*mandatory to be set*	Credential location. In case of *jks*, *pkcs12* and *pem* store it is the only location required. In case when credential is provided in two files, it is the certificate file path.

Property name	Type	Default value / mandatory	Description
`xuudb.credential.format`	[jks, pkcs12, der, pem]	-	Format of the credential. It is guessed when not given. Note that *pem* might be either a PEM keystore with certificates and keys (in PEM format) or a pair of PEM files (one with certificate and second with private key).
`xuudb.credential.password`	string	-	Password required to load the credential.
`xuudb.credential.keyPath`	string	-	Location of the private key if stored separately from the main credential (applicable for *pem* and *der* types only),
`xuudb.credential.keyPassword`	string	-	Private key password, which might be needed only for *jks* or *pkcs12*, if key is encrypted with different password then the main credential password.
`xuudb.credential.keyAlias`	string	-	Keystore alias of the key entry to be used. Can be ignored if the keystore contains only one key entry. Only applicable for *jks* and *pkcs12*.

Examples

Note
Various UNICORE modules use different property prefixes. Here we don't put any, but in practice you have to use the prefix (see the reference table above for the actual prefix). Also properties might need to be provided using different syntax, as XML.

Credential as a pair of DER files:

```
credential.format=der
credential.password=the\!njs
credential.path=/etc/credentials/cert-1.der
```

```
credential.keyPath=/etc/credentials/pk-1.der
```

Credential as a JKS file (credential type can be autodetected in almost every case):

```
credential.path=/etc/credentials/server1.jks
credential.password=xxxxxx
```

3.4 Dynamic mappings configuration

Dynamic mappings are configured with a set of rules. When designing rules it is good to remember that all users, which will be evaluated, were already successfully authorized.

Each rule has a condition which selects users and a list of mappings which should be applied for the selected users. Example conditions (in English):

- all members of a */vo.wonderworld.gov*
- all (authorized) users
- all users having extra attribute *matlabAllowed* with any value AND being member of a subgroup of */vo.wonderworld.gov/dynamic/*

Example mappings (in English):

- add user a supplementary group *matlab*
- assign uid from a pool of existing uids
- assign a fixed gid *grid*
- invoke an external program and use its standard output as users gid

Precisely speaking, a mapping must have defined:

- what attribute it maps: *uid*, (primary)*gid* or *supplementaryGids*,
- using what method: *fixed*, *pool* or *script*

Additionally one can define an optional parameter stating if the mapping should overwrite an attribute value which was previously set (either by an earlier rule or assigned using a different attribute source).

As it was mentioned there are three kinds of mappings. Let's shortly introduce them one by one.

3.4.1 Fixed mappings

Fixed mappings are the most basic option. The mapping is formed by a simple assignment of a fixed value. It can be used to:

- assign a common (shared!) uid to selected users (rarely used)
- assign a fixed gid to selected users (very useful to assign a gid to all Grid users, or all members of a VO)
- assign some supplementary gids to selected users (useful to provide additional local permissions to users having a special role/attributes/etc.).

The example in the pool mappings section contains also a fixed mapping.

3.4.2 Script mappings

Script mappings are the "Do It Yourself" mechanism. You can provide a command line which will be parsed and invoked. The application must return (on its standard output) a string with a mapping result (depending on what is mapped - gid, uid or a space separated list of supplementary gids). Of course the script can be informed who is actually being mapped, by using parameters enclosed in `${}`. The list of available parameters is given below.

- `userDN` user's DN
- `issuerDN` user's certificate issuer's DN
- `role` user's role
- `vo` user's selected VO
- `extraAttributes` map with extra attributes, names are the keys
- `xlogin` user's uid (if already established)
- `gid` user's gid (if already established)
- `supplementaryGids` user's supplementary gids (if already established)
- `xloginSet` whether uid was set
- `gidSet` whether gid was set
- `dryRun` whether the current invocation is only a simulation, and shouldn't affect any persisted system settings.

The example below contains also a script mapping.

3.4.3 Pool mappings

Finally the *pool mappings* are both flexible and relatively easy to use --- it is the most advanced mapping type. Using the pool mapping you have to prepare a set of reserved identifiers (uids or gids depending on what is mapped). The related system accounts can be precreated or can be created on-demand. The pool mapping is configured with an additional, very important parameter: pool key. Pool key is a name of one of the user's attributes: *userDN*, *issuerDN* (DN of CA which issued user's certificate), *role*, *vo* or any other generic user's attribute.

To explain how the pool works let's assume that key is set to *userDN*. Then the pool will map a user as follows: first it is checked if there is an existing mapping bound to the user's DN. If it is found then it is simply returned. If not (the user is trying to use the site for the first time) a new identifier is selected from the pool, and stored under the key being the user's DN. Then the new identifier is returned.

Therefore all users having the same value of the pool key will get the same mapping and vice versa. If DN is the key then all users will have a distinct mapping (useful for uids or for gids, if every user should get a unique one). If for instance a VO is the key then all VO members will have the same mapping (useful for gid, or for uid if all VO members should have the same user account).

The following example should help to understand those concepts and is also providing a basic syntax reference:

```
<?xml version="1.0" encoding="UTF-8"?>
<dynamicAttributes xmlns="http://unicore.eu/xuudb/ ↩
    dynamicAttributesRules"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <rules>
    <!-- all members of the vo /vo.wonderworld.gov should have a  ↩
        common uid 'shared_user'-->
    <rule>
      <condition>vo.matches("/vo.wonderworld.gov/.*")</condition>
      <mapping type="fixed" maps="uid">shared_user</mapping>
    </rule>

    <!-- all users with a role 'admin' should get a primary gid  ↩
        from the 'admins-pool' pool.
    For pools the 'maps' parameter is optional - it is better to  ↩
        specify it in the pool definition,
    below. -->
    <rule>
      <condition>role="admin"</condition>
      <mapping type="pool">admins-pool</mapping>
    </rule>

    <!-- all users from the /biology VO get an uid from the pool  ↩
        and a fixed primary gid 'biol' -->
    <rule>
```

```
      <condition>vo.matches("/biology/.*")</condition>
      <mapping type="pool">biology-uids-pool</mapping>
      <mapping type="fixed" maps="gid">biol</mapping>
    </rule>

    <!-- complicated condition: all users who have a generic  ↩
        attribute 'matlabAllowed' set AND the value
        of this attribute is 'true' get a supplementary group ' ↩
            matlab' -->
    <rule>
      <condition>attributes["matlabAllowed"] != null and attributes ↩
          ["matlabAllowed"].contains("true")</condition>
      <mapping type="fixed" maps="supplementaryGids">matlab</ ↩
          mapping>
    </rule>

    <!-- all (authorized) users, who do not have an uid set ( ↩
        overwriteExisting=false)
        should have an uid assigned by a script /usr/local/bin/ ↩
            create-mapping.pl. The script will be called
        with two arguments: user's DN and VO.
    <rule>
      <condition>true</condition>
      <mapping type="script" maps="uid" overwriteExisting="false">/ ↩
          usr/local/bin/create-mapping.pl "${userDN}" "${vo}" </ ↩
          mapping>
    </rule>
  </rules>

  <!-- Here come pools -->
  <pools>
    <!-- pool 'admins-pool' maps gids. The list of gids provides  ↩
        groups which were
    pre-created in the system. The gids will be stored per-user dn, ↩
         so every admin will get another group.
    Finally the list of gids uses special expressions where number  ↩
        ranges are provided.
    -->
    <pool id="admins-pool" type="gid" key="dn" precreated="true">
      <id>admin_grp[1-100]</id>
      <id>admin_grp[200-1000]</id>
    </pool>
    <!-- This pool identifiers are loaded from an external file -->
    <pool id="biology-uids-pool" type="uid" key="dn" precreated=" ↩
        true">
      <file>src/test/resources/externalUidsPool</file>
    </pool>
  </pools>
</dynamicAttributes>
```

Usage of pools brings several issues regarding old mappings removal and notifications about pools getting empty. In the first case it suggested not to remove the users for the time a VO or Grid is supported: it is a simplest approach, and nowadays operating systems can support thousands of uids without any problem (Linux can have 32bit uid numbers).

In case a site wants to recycle mappings, XUUDB offers the following mechanism:

- Inactive mappings can be automatically (after a configurable time threshold) or manually (using the admin client) *frozen*. An identifier belonging to a frozen mapping is still assumed to be occupied, but the mapped user won't have it assigned (in the unprobable case that she returns to the site). Freezing is introduced to give a time for tidying up local resources assigned to the identifier. Such cleaning must be done manually and should include removal of all owned files and killing any processes. Of course this depends whether the identifier was a gid or uid. Also please note that in case of clusters, all nodes should be cleaned up.
- After the clean up is done, the frozen mapping can be removed, again manually using the admin client or automatically, after staying in the frozen state for a specified amount of time. Note that it is impossible to remove an alive mapping.

If administrator is able to provide scripts which performs cleanup, then it is possible to invoke them upon pool mapping freezing and automate the whole process. In a similar way other handlers may be configured and XUUDB will invoke them to notify about mappings removal, assignment of a new mapping (useful when accounts are not pre-created but should be created on demand) and also when a pool is getting empty.

The following example shows all the possible handlers and lists arguments which are passed to them. As it can be seen all pool options including handlers, can be configured globally or per-pool.

```
<?xml version="1.0" encoding="UTF-8"?>
<dynamicAttributes xmlns="http://unicore.eu/xuudb/ ↩
    dynamicAttributesRules"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- how often (in s) pools should be checked for old or inactive ↩
      mappings -->
  <poolMonitoringDelay>300</poolMonitoringDelay>
  <defaultConfiguration>
        <!-- in seconds: automatic freezeing (time measured from  ↩
            last mapping use)... -->
        <automaticFreezeAfter>3600000</automaticFreezeAfter>
        <!-- ... and final removal (time measurd from mapping  ↩
            freeze) -->
        <automaticDeleteAfter>36000</automaticDeleteAfter>
        <!-- when less then this free mappings are left generate a  ↩
            warning -->
        <emptyWarningAbsolute>20</emptyWarningAbsolute>
        <!-- when less then this percent of free mappings is left  ↩
            generate a warning -->
        <emptyWarningPercent>5</emptyWarningPercent>
```

```
    <!-- timeout for running ALL external programms -->
    <handlerInvocationTimeLimit>10000</ ↩
        handlerInvocationTimeLimit>

    <!-- Various handlers. Arguments are pool.getId(), pool. ↩
        getType().toString(),
                                    bean.getEntry(), oldSec+""  ↩
                                        -->

    <!-- Handler invoked before freezing an account.
Arguments: <poolId> <poolType> <identifier> < ↩
     inactiveForInSeconds>
If handler returns a non-zero exit status then the freezing is ↩
      skipped
(unless invoked by admin-client).
    -->
    <handlerAboutToFreeze>/opt/handlers/releaseAccountResources ↩
        .sh</handlerAboutToFreeze>

    <!-- Handler invoked before deleting a frozen identifier.
         Arguments: <poolId> <poolType> <identifier> < ↩
             frozenForInSeconds>
         If handler returns a non-zero exit status then the  ↩
             deletion is skipped
         (unless invoked by admin-client).
         -->
    <handlerAboutToDelete>/opt/handlers/notifyAccountRecycled. ↩
        sh</handlerAboutToDelete>

    <!-- Handler invoked when an identified from the uids pool  ↩
        is going to be used for the first time
    (or for the first time after deleting it), if the pool is  ↩
        set as not pre-created.
            Arguments: <poolId> <uid> <key>
    -->
    <handlerCreateSystemUid>/opt/handlers/adduser.sh</ ↩
        handlerCreateSystemUid>

    <!-- Handler invoked when an identified from the gids pool  ↩
        is going to be used for the first time
    (or for the first time after deleting it), if the pool is  ↩
        set as not pre-created.
            Arguments: <poolId> <gid> <key>
    -->
    <handlerCreateSystemGid>/opt/handlers/addgroup.sh</ ↩
        handlerCreateSystemGid>

    <!-- Handler invoked when a pool warning threshold is  ↩
        exceeded.
```

```
             Arguments: <poolId> <poolType> <remainingFreeIds>
       -->
       <handlerPoolGettingEmpty>/opt/handlers/notifyNearlyEmpty.sh ↩
           </handlerPoolGettingEmpty>

       <!-- Handler invoked when a pool gets empty.
            Arguments: <poolId> <poolType>
       -->
       <handlerPoolEmpty>/opt/handlers/notifyEmpty.sh</ ↩
           handlerPoolEmpty>
  </defaultConfiguration>

  <rules>
       <!-- some rules ........ -->
  </rules>

  <pools>
       <!-- Pool can overwrite any of the global configuration  ↩
           options -->
    <pool id="admins-pool" type="gid" key="dn" precreated="true">
      <configuration>
         <!-- disable automatic freezing for this pool -->
                 <automaticFreezeAfter>-1</automaticFreezeAfter>
      </configuration>
      <id>admin_grp[1-100]</id>
      <id>admin_grp[200-1000]</id>
    </pool>
  </pools>
</dynamicAttributes>
```

3.5 Starting the XUUDB server

Start the server with

```
BIN/start.sh
```

In case if XUUDB was installed with binary package use:

```
/etc/init.d/unicore-xuudb start
```

3.6 Stopping the server

Stop the server with

```
BIN/stop.sh
```

This sends a TERM signal to the XUUDB process. Please do not use `kill -9` to stop XUUDB, to avoid corrupting the database.

In case if XUUDB was installed with binary package use:

```
/etc/init.d/unicore-xuudb stop
```

3.7 Logging

UNICORE uses the Log4j logging framework. It is configured using a config file. By default, this file is found in components configuration directory and is named `logging.properties`. The config file is specified with a Java property `log4j.configuration` (which is set in startup script).

Several libraries used by UNICORE also use the Java utils logging facility (the output is two-lines per log entry). For convenience its configuration is also controlled in the same `logging.properties` file and is directed to the same destination as the main Log4j output.

Note
You can change the logging configuration at runtime by editing the logging.properties file. The new configuration will take effect a few seconds after the file has been modified.

By default, log files are written to the the `LOGS` directory.

The following example config file configures logging so that log files are rotated daily.

```
# Set root logger level to INFO and its only appender to A1.
log4j.rootLogger=INFO, A1

# A1 is set to be a rolling file appender with default params
log4j.appender.A1=org.apache.log4j.DailyRollingFileAppender
log4j.appender.A1.File=logs/uas.log

#configure daily rollover: once per day the uas.log will be copied
#to a file named e.g. uas.log.2008-12-24
log4j.appender.A1.DatePattern='.'yyyy-MM-dd

# A1 uses the PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c{1} %x -  ↩
    %m%n
```

Note
In Log4j, the log rotation frequency is controlled by the DatePattern. Check http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/DailyRollingFileAppender.html for the details.

For more info on controlling the logging we refer to the log4j documentation:

- PatternLayout
- RollingFileAppender
- DailyRollingFileAppender

Log4j supports a very wide range of logging options, such as date based or size based file rollover, logging different things to different files and much more. For full information on Log4j we refer to the publicly available documentation, for example the Log4j manual.

3.7.1 Logger categories, names and levels

Logger names are hierarchical. In UNICORE, prefixes are used (e.g. "unicore.security") to which the Java class name is appended. For example, the XUUDB connector in UNICORE/X logs to the "unicore.security.XUUDBAuthoriser" logger.

Therefore the logging output produced can be controlled in a fine-grained manner. Log levels in Log4j are (in increasing level of severity):

`TRACE` on this level *huge* pieces of unprocessed information are dumped, # `DEBUG` on this level UNICORE logs (hopefully) admin-friendly, verbose information, useful for hunting problems, # `INFO` standard information, not much output, # `WARN` warnings are logged when something went wrong (so it should be investigated), but recovery was possible, # `ERROR` something went wrong and operation probably failed, # `FATAL` something went really wrong - this is used very rarely for critical situations like server failure.

For example, to debug a security problem in the UNICORE security layer, you can set:

```
log4j.logger.unicore.security=DEBUG
```

If you are just interested in details of credentials handling, but not everything related to security you can use the following:

```
log4j.logger.unicore.security=INFO
log4j.logger.unicore.security.CredentialProperties=DEBUG
```

so the XUUDBAuthoriser will log on DEBUG level, while the other security components log on INFO level.

Note

(so the full category is printed) and turn on the general DEBUG logging for a while (on unicore). Then interesting events can be seen and subsequently the logging configuration can be fine tuned to only show them.

Several logging categories common in XUUDB:

Log category	Description
unicore	All of UNICORE
unicore.security	Security layer
unicore.client	Client calls (to other servers)
unicore.xuudb	XUUDB related
unicore.xuudb.server	XUUDB server
unicore.xuudb.server.db	XUUDB server database layer
unicore.xuudb.client	XUUDB admin client

4 The admin client

The admin client is used to edit the XUUDB, using a web service interface. It is configured in the file `CONF/xuudb_client.conf`. Client is invoked using the following pattern:

```
ADMIN <command> <options>
```

You can get detailed usage info by calling the admin script without any options. As it was noted above the actual utility path is dependent on how XUUDB was installed: it is either `/usr/sbin/unicore-xuudb-admin` or `INST/bin/admin.sh`.

Note
to switch on the confirmation message asked by the `add` command, edit the admin.sh script, so that the `xuudb.batch` property is set to `false`.

The client configuration requires the URL of the XUUDB server in the property `xuudb.address` and in case of secure HTTPS connections also a configuration truststore and credential. The settings are exactly the same as in case of the XUUDB server, so refer to its documentation: Section 3.3.4.

4.1 Configuring advanced HTTP client settings

UNICORE client stack can be configured with several advanced options. In most cases you can skip this section as defaults are fine.

The following table lists all available options. A special note for the *http.** properties: those are passed to the Apache Commons HTTP Client library. Therefore it is possible to configure all relevant options of the client. The options are listed under this address: http://hc.apache.org/-httpclient-3.x/preference-api.html Also see the example below.

Property name	Type	Default value / mandatory	Description
`xuudb.client.digitalSigningEnabled`	[true, false]	`true`	Controls whether signing of key web service requests should be performed.
`xuudb.client.httpAuthEnabled`	[true, false]	`false`	Whether HTTP basic authentication should be used.
`xuudb.client.httpPassword`	string	*empty string*	Password for use with HTTP basic authentication (if enabled).
`xuudb.client.httpUsername`	string	*empty string*	Username for use with HTTP basic authentication (if enabled).
`xuudb.client.inHandlers`	string	*empty string*	Space separated list of additional handler class names for handling incoming WS messages
`xuudb.client.maxWsCallRetryNumber`	integer number	`3`	Controls how many times the client should try to call a failing web service. Note that only the transient failure reasons cause the retry. Note that value of 0 enables unlimited number of retries, while value of 1 means that only one call is tried.
`xuudb.client.messageLogging`	[true, false]	`false`	Controls whether messages should be logged (at INFO level).
`xuudb.client.outHandlers`	string	*empty string*	Space separated list of additional handler class names for handling outgoing WS messages
`xuudb.client.securitySessions`	[true, false]	`true`	Controls whether security sessions should be enabled.

Property name	Type	Default value / mandatory	Description
xuudb.client.serverHostnameChecking	[NONE, WARN, FAIL]	WARN	Controls whether server's hostname should be checked for matching its certificate subject. This verification prevents man-in-the-middle attacks. If enabled WARN will only print warning in log, FAIL will close the connection.
xuudb.client.sslAuthEnabled	[true, false]	true	Controls whether SSL authentication of the client should be performed.
xuudb.client.sslEnabled	[true, false]	true	Controls whether the SSL/TLS connection mode is enabled.
xuudb.client.wsCallRetryDelay	integer number	10000	Amount of milliseconds to wait before retry of a failed web service call.
--- HTTP client settings ---			
xuudb.client.http.chunkingDisabled	[true, false]	true	If set to false, then the client will not use HTTP 1.1 data chunking.
xuudb.client.http.connection-close	[true, false]	false	If set to true then the client will send connection close header, so the server will close the socket.
xuudb.client.http.connectionTimeout	integer number	20000	Timeout for the connection establishing (ms)
xuudb.client.http.maxConnectionsPerHost	integer number	6	How many connections per host can be made. Note: this is a limit for a single client object instance.
xuudb.client.http.maxRedirects	integer number	3	Maximum number of allowed HTTP redirects.
xuudb.client.http.maxTotalConnections	integer number	20	How many connections in total can be made. Note: this is a limit for a single client object instance.
xuudb.client.http.socketTimeout	integer number	0	Socket timeout (ms)
--- HTTP proxy settings ---			

Property name	Type	Default value / mandatory	Description
`xuudb.client.http.nonProxyHosts`	string	-	Space (single) separated list of hosts, for which the HTTP proxy should not be used.
`xuudb.client.http.proxy.password`	string	-	Relevant only when using HTTP proxy: defines password for authentication to the proxy.
`xuudb.client.http.proxy.user`	string	-	Relevant only when using HTTP proxy: defines username for authentication to the proxy.
`xuudb.client.http.proxyHost`	string	-	If set then the HTTP proxy will be used, with this hostname.
`xuudb.client.http.proxyPort`	integer number	-	HTTP proxy port. If not defined then system property is consulted, and as a final fallback 80 is used.
`xuudb.client.http.proxyType`	string	`HTTP`	HTTP proxy type: HTTP or SOCKS.

Example

> **Note**
> Various UNICORE modules use different property prefixes. Here we don't put any, but in practice you have to use the prefix (see the reference table above for the actual prefix). Also properties might need to be provided using different syntax, as XML.

Here we are setting an extremely short connection and socket timeouts for the clients calls, using the Apache HTTP client parameters. Additionally server hostname to certificate subject name checking is set to cause connections failures, preventing man in the middle attacks.

```
client.http.connection.timeout=2000
client.http.socket.timeout=2000
client.serverHostnameChecking=FAIL
```

4.2 Commands

The help with examples is provided for each command. You can use `helpAll` to print a reference documentation for all commands. Selected commands are also described below.

```
[classic]
  add
  adddn
  check-cert (chc)
  check-dn (chdn)
  export
  import
  list
  remove
  update

[dynamic]
  findMapping (fm)
  findReverseMapping (fr)
  freezeMappings
  getDynamicAttributes (getDyn)
  listMappings (lm)
  listPools (lp)
  removeMappings
  removePool (rmp)
  simulate (sim)

[other]
  help
  helpAll
```

Common options:

gcID
: The so-called "grid component ID" is used to group entries, and must match the setting in the UNICORE/X configuration file `uas.config`. For example if you have two systems with different user name mappings, you can handle both with a single XUUDB, since you can store two user name mappings for each certificate, by choosing a different gcID for both systems. When updating xuudb entries, the special gcid * can be used as wildcard for updating user entries on all systems.

pemfile
: A file containing a public key in PEM format

DN
: The distinguished name of a user

xlogin
: xlogins (from UNIX login) are used for incarnation. Grid user's request which results in

invocation of operations on a target system (usually through BSS) must be mapped to a local UNIX user. This attribute specifies the XLogins which are valid for the user. The first one is also used as a default one, if user does not request a particular one. Multiple logins can be specified using a `:`

project
Defines a primary group UNIX group for a user. If it is undefined then a default group for the XLogin is used.

role
The usual roles in UNICORE are `user` for a normal user, and `admin` for an administrator. Custom roles can be added, and can be assigned permissions in the UNICORE/X security policy file.

4.3 Adding entries using `add` or `adddn`

Example using the DN:

```
$> ADMIN adddn DEMO-SITE "CN=John Doe, O=Test Inc" userlogin ↩
    user
```

Example using a pem file:

```
$> ADMIN add DEMO-SITE /path/to/usercert.pem userlogin user
```

Note
When extracting the DN from a certificate file using OpenSSL, make sure to use the `RFC2253` option, for example:

```
openssl x509 -in demouser.pem -noout -subject -nameopt  ↩
   RFC2253
```

4.4 Checking the content

Apart from `list`, you can use the `check-cert` and `check-dn` commands to see what the XUUDB contains for a certain certificate or DN.

4.5 Removing entries

HINT: before removing you can check with the list command which takes the same parameters, that your are removing the correct entries.

To remove all entries from xuudb (you will have to confirm this)

```
$> ADMIN remove ALL
```

To remove some entries, you have to specify attributes.

To remove a user with cert cert.pem at gcid MYSITE:

```
$> ADMIN remove gcid=id001 pemfile=/path/cert.pem
```

To remove all users from gcid OLDMACHINE:

```
$> ADMIN remove gcid=OLDMACHINE
```

To remove a user with xlogin jdoe from all gcids:

```
$> ADMIN remove xlogin=jdoe
```

etc...

4.6 Exporting/importing

The export command creates a csv file, which will contain the complete XUUDB database:

```
$> ADMIN export uudb.csv
```

If the file already exists, the export tool will complain. To override this, please specify the `overwrite` option, e.g.

```
$> ADMIN export uudb.csv overwrite
```

The import command takes the a csv file (as generated by `export`) and imports all entries. Already existing entries will not be changed. To do updates, execute `admin.sh remove ALL` before, or specify `clearDB` as a second argument

```
$> ADMIN import uudb.csv
```

4.7 Updating entries

The `update` command can be used to modify existing entries, for example to replace the certificate or the login. For example,

```
$> ADMIN update DEMO-SITE certs/demouser.pem xlogin=jb007
```

would update the entry identified by the gcID *DEMO-SITE* and the given pem file, and assign a new xlogin. If you want to update a user's entry on all the sites, you would use

```
$> ADMIN update \* certs/demouser.pem xlogin=jb007
```

Note that the wildcard * is a special character for the shell and needs to be escaped with a backslash.