EGEE

EGEE gLite Metadata Catalog User's Guide

gLite Metadata Catalog Interface Description

Document identifier:	**EGEE-TECH-573725-v1.2**
Date:	**April 12, 2005**
Activity:	**JRA1: Data Management**
Document status:	**DRAFT**
Document link:	**https://edms.cern.ch/document/573725**

Abstract: This is the user's guide to the gLite Metadata Catalog. The basic concepts are defined here as well as all of the methods in the interfaces. The description is language-neutral. The language-specific guides with examples are provided as separate guides. The aim of this guide is to explain the design, concepts and possible applications.

Document Change Log

Issue	Date	Comment	Author
1.0	1.3.2005	Initial Version.	Text by Ricardo Rocha and Peter Kunsztt
1.1	31.3.2005	Updated based on PTF recommendations.	Peter Kunsztt
1.2	11.4.2005	Working out details, esp. MQL.	Ricardo Rocha

Document Change Record

Issue	Item	Reason for Change

CONTENTS

1. INTRODUCTION

Metadata is in general a notion of 'data about data'. There are many aspects of metadata, like descriptive metadata, provenance metadata, historical metadata, security metadata, etc. Due to this multi-faceted nature of metadata, it is very difficult to define an interface which will suit all possible applications.

We base the interface describe in this document on experience by the usage of the Replica Metadata Catalog in the EU DataGrid and the LHC Computing Grid projects, on the usage of metadata in the AliEn File Catalog, on the definition and prototype implementation of the ARDA Metadata interface [1]. In addition, there are a set of use cases and requirements coming from the High Energy Physics and Biomedical application domains which are recorded in the EGEE Project Technical Forum database [7]. See also the HEP Metadata Group Use Cases document [6] (and references therein).

1.1. BASIC CONCEPTS

We use some concepts in the Metadata interface described here that are common across most use cases. In order to discuss these concepts we need to define our terminology:

Entry A metadata catalog entry consists of

Key A string that contains the identifier by which the metadata entry will be referenced. This key must be unique.

Attributes Each attribute may hold information (metadata) about the entry.

In the interface we assume the key to be a string - a file name (like an LFN or GUID, see [5]), a job identifyer, a patient name, etc; whatever the 'thing' is to which metadata is to be assigned. Of course implementations may choose to store the key as another type in the backend.

Attributes An attribute has:

- a **schema** – a string that identifies the schema to which the attribute belongs to within the catalog.
- a **name** – a string that specifies the attribute name within its schema. Attribute names must be unique within the same schema.
- a **value** – represented/encoded as a string.
- a **type** – a string indicating what kind of information is contained in the value. Different metadata implementations may accept a different set of types.

Schema Metadata is associated with entries in the catalog via schemas, which define groups of attributes. Schemas can be thought of as tables in a relational database where the attributes define the columns. There may be many schemas defined in the Metadata Catalog. A schema has:

- a **name**, which is a string that must be unique (no two schemas may have the same name within the Metadata Catalog).
- a **list of attributes**, which defines the schema.

Each catalog entry must have at least one associated schema. Each attribute must be part of a schema. An attribute is uniquely defined within the Metadata Catalog by its name and the schema name that it belongs to.

Permission A permission in gLite consists of

- a BasicPermission, which stores a username, groupname and permission numbers for the user, group members and others.
- a list of ACLs. The ACL in gLite consists of a principal name (which can be any string, so either username or group name) and a list of access control bits for read, write, execute, list, remove, permission, getMetadata and setMetadata.

Permissions are applied at the entry and schema level in the catalog. This provides enough granularity for most applications, but is not enough where attribute-level security is needed.

Policy Policies in gLite consist of a string containing a WHERE clause as defined in the MQL query language. They provide a way to restrict the access to entries in the catalog based on attribute values.

MQL The Metadata Query Language. This is an internal definition of a query language to access metadata inside a catalog following the interface described in this document. It consists of a subset of all the SQL standard functionality, expressed in 'SELECT ... WHERE ...' format strings. For the grammar definition and some examples see section 4..

The current gLite Metadata Catalog functionality is spread over four interfaces. These are described in the quickstart and reference sections in detail.

MetadataBase Two sets of operations are offered through this interface:

- Querying and setting values of attributes for individual entries/items in the catalog.
- Generic queries returning entry/item identifiers.

MetadataCatalog The necessary operations for managing entries in the catalog are available through this interface. This includes functionality for creating and deleting items.

MetadataSchema The necessary operations for handling schemas inside the catalog.

FASBase The set and get permission methods are inherited in MetadataBase from this interface.

An implementation of a Metadata Catalog can choose not to have all the functionality defined in these interfaces. A concrete example is a File Catalog that may also want to offer file metadata. Functionality for managing entries/items in the catalog is already provided as part of the File Catalog, so the MetadataCatalog interface is not needed. The MetadataBase interface would be enough to have POSIX xattrs functionality, and it can be extended with the MetadataSchema interface to add schema management if desired (as is the case in the gLite Fireman catalog [4]).

1.2. COMMON IMPLEMENTATION OBJECTS

To guarantee interoperability an implementation must closely follow the interface definition of a Metadata Catalog. Additionaly, it must provide pre-defined objects which may be used within queries to the catalog.

request This is the first of these objects. It is a virtual schema, providing information about the current request. An implementation must expose the following items inside this schema:

clientDN A string containing the Distinguish Name of the client, taken from her certificate.

clientVOMSAttributes A string containing all the attributes exposed by the client, taken from the VOMS credential. They should be put in a string in the form 'attr1,attr2,...,attrN'. This format allows ease of use inside 'schemaName.attrName IN request.clientVOMSAttributes' expressions in queries.

entry This is the second object. It is a real schema, one that can be linked with entries and have metadata. In this case, all entries will be linked with this schema by default (without need of an explicit action), giving default attributes for every entry. The following items are exposed inside the schema:

ID Holds the unique identifier of the entry in the catalog. It may be used within queries to retrieve the identifier of entries, or it can be used inside a setAttributes call to rename the entry in the catalog.

1.3. Additional Concepts

Some concepts that are common in applications dealing with metadata are not explicitly used in this document. These concepts can be mapped on the basic concepts described above. The design of the interface described here was driven by the principle of keeping it as simple as possible, while allowing most applications to use the catalog to implement their additional concepts.

Logical Dataset Just like Logical Filenames (described in [5]), logical datasets are simply entries in the metadata catalog. Logical Datasets are described in the HEPCAL use case document [2, 3].

Collection In many applications data entries are grouped together into collections. The collections may have different semantics from application to application (having set or group semantics for example) but the few most basic operations (add and remove member) remain universal. The collections map almost exactly on the schema concept we have described in the previous section, since entries in a collection have the same types of attributes. If several collections are needed that share the very same schema, this may be implemented by adding an additional 'collection' attribute to the schema which then may be used in queries as an additional constraint on selection of entries and attributes.

Dataset Another common concept is that of datasets. The difference between datasets and collections is that entries in a collection share the same schema while datasets are simply groups of entries without the requirement of sharing the same schema. Datasets may be hierarchical, i.e. a dataset may contain another dataset. Since this concept does not map on the actual metadata (other than datasets themselves may have metadata), there is no support for this in the Metadata Catalog we define here. Datasets may be entries in the Metadata Catalog, so it can store their metadata but the hierarchy and grouping concepts are a separete set of functionality which would need an additional different interface to support. For some applications, the hierarchical structure available in the File Catalog may be sufficient to implement datasets.

Virtual data Some applications virtualize their data by computing it on demand, storing only the metadata necessary to recreate it. Such metadata may be defined in the Metadata Catalog as described here by choosing the appropriate attributes. So a virtual data catalog may be implemented in a straightforward way on top of this interface.

Virtual Dataset A virtual dataset is sometimes defined as the result of a query in the metadata catalog. The result may vary, depending on the content of the catalog. The Metadata Catalog as defined here may be used to store such queries as an attribute (provided the varchar type is long enough to hold the query string). The query may be retrieved and re-executed anytime by the application, thus giving it the virutal dataset capability if needed.

1.4. INTERACTIONS WITH OTHER SERVICES

The Metadata Catalog is a standalone service that others may depend on but it does not depend on any other grid service (except for the usual security mechanism dependencies, e.g. VOMS).

For an explanation of how the metadata catalog fits into the whole set of data management components in gLite, see also the Overview of gLite Data Management User's Guide [5].

2. REFERENCE GUIDE

In this section we describe each interface and their methods in detail, focussing on their semantics. The interface inheritance diagram 1 shows the interfaces that can be used when interacting with a Metadata Catalog.

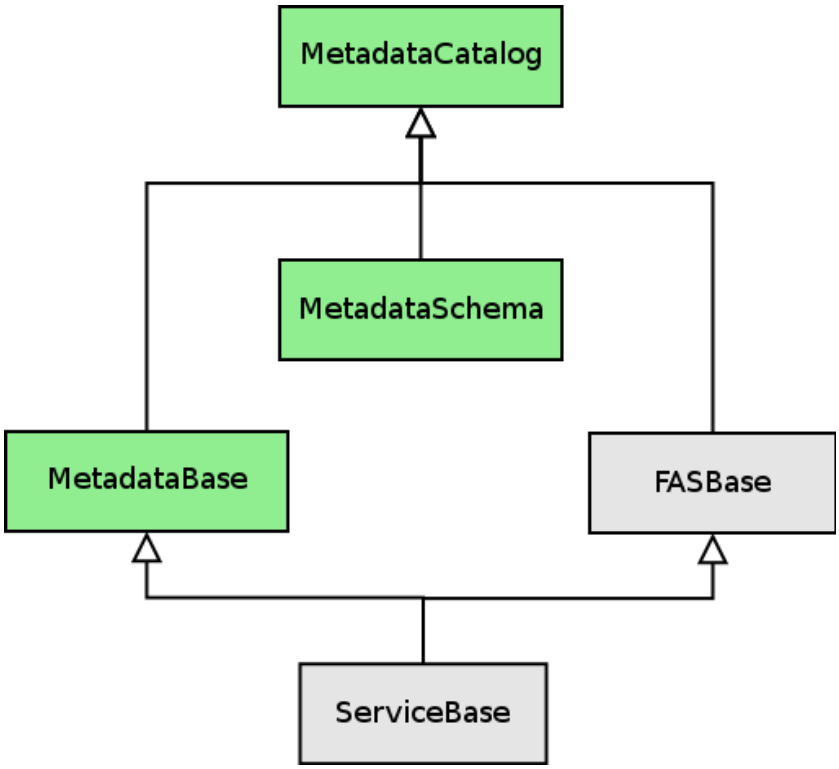

Figure 1: The Metadata Interface inheritance. The Base classes include generic methods, while the MetadataSchema and MetadataCatalog classes contain the metadata-specific operations.

2.1. BASE OPERATIONS

This section defines the `MetadataBase` interface. The Operations in this interface are the following:

`listAttributes` List all attributes associated with an entry.

`setAttributes` Sets the values for a group of attributes of entries.

`query` Performs a query on the catalog, returning the requested items and/or attributes.

`nextQuery` Continue with a query, retrieving the next batch of results.

`endQuery` Notify the server that the query will not be continued from the client anymore.

There are many auxiliary objects used in these methods. For a detailed description see Section 2.6.

2.1.1. LISTATTRIBUTES

This method lists all attributes associated with an entry. The `entry` passed in the request is the unique identifier of the entry in the catalog.

```
Attribute[] listAttributes(String entry)
```

The returned list of `Attribute` objects (see section 2.6.1.) contains the name of the schema the attribute belongs to, the attribute name and its type. All attributes associated with the entry and visible by the client MUST be returned - including the ones with no actual value set for the entry.

Return Value: An array of `Attribute` objects. Returns null if the entry has no attributes associated (ie it has not been associated to a schema yet).

Errors

`AuthorizationException` No access right to access attributes for this entry.

`NotExistsException` The entry specified does not exist.

`InternalException` Any other error on the server side (i.e. database down).

2.1.2. SETATTRIBUTES

```
int setAttributes(String query, Attribute[] attributes)
```

This method sets the values for a group of attributes of entries. The `query` string contains a partial MQL query (only the WHERE clause part). It is a restriction defining the entries that should be affected by the operation. The `attributes` parameter is a list (array) of `Attribute` objects (see section 2.6.1., each containing the necessary information to set the new values (`schema/name/value` triplet).

See Section 4. for detailed instructions on how to write your query.

Return Value: The number of entries in the catalog that were affected by the operation.

Errors

`AuthorizationException` No access right to update values of attributes.

`NotExistsException` One of the attributes specified does not exist or one of the entries specified in the query does not exist.

`InvalidArgumentException` One of the `name/schema` pairs given for an attribute is invalid; or one of the attribute's values is invalid while trying to update values; or the query passed is invalid.

`InternalException` Any other error on the server side (i.e. database down).

2.1.3. QUERY

```
MDResult query(MDQuery query)
```

This method performs a generic query on the catalog, returning the requested attribute values. It might return all the results in case of a small resultset, or it might return only partial results if the query results are above the size defined by the server. To iterate through results, further calls should be made to **??**.

The query is given using a `MDQuery` object which is described in Section 2.6.3.

Return Value: The resulting `MDResult` object is described in Section 2.6.4. If its boolean `done` field is true, all of the results have been placed in the `attributes` field. Otherwise, the `nextQuery` method has to be invoked to get the next batch of results for the given query from the server, passing it the `token` field of the `MDResult` object retrieved by the previous query.

Errors

`AuthorizationException` No access right to list the entries.

`NotExistsException` Some of the requested attributes do not exist.

`InvalidArgumentException` The requested attribute contains invalid data.

`InvalidQueryException` The query is invalid in some way.

`InternalException` Any other error on the server side (i.e. database down).

2.1.4. NEXTQUERY

```
MDResult nextQuery(String token, MDQuery query)
```

This method retrieves the next set of results from the server for a given query. The `token` field has to be the one from the `MDResult` object retrieved by the previous query. The query is repeated in the method signature in order to enable stateless implementations of the Metadata Catalog. For some implementations (which keep state on the server) the `query` parameter of this method may be left empty.

Return Value: The resulting `MDResult` object is described in Section 2.6.4. If its boolean `done` field is true, all of the remaining results have been placed in the `attributes` field. Otherwise, the method has to be invoked again to get the next batch of results for the given query from the server. The amount of data placed into the `attributes` field is determined by the server's capabilities and maybe by some client configuration.

Note: See also discussion in Section 2.1.6.

Errors

`AuthorizationException` No access right to perform the query.

`NotExistsException` Some of the requested attributes do not exist.

`InvalidArgumentException` One of the parameters passed to the method is invalid.

`InvalidQueryException` The query is invalid in some way, the token is out of scope, has expired or it has already returned in a previous call with the done field set to true.

`InternalException` Any other error on the server side (i.e. database down).

```
void endQuery(String token)
```

2.1.5. ENDQUERY

In stateful implementations of the interface, this method may be called by the client to inform the server that a given token is not needed any longer and that its resources may be freed. This call is without effect on stateless implementations of the interface.

Errors

`InvalidArgumentException` The token is invalid.

`InvalidQueryException` The query the token refers to is invalid in some way, the token has expired or has been already closed by a previous call to this method or has completed already.

`InternalException` Any other error on the server side (i.e. database down).

2.1.6. CLIENT-SIDE METHOD WRAPPERS

We suggest that the query methods are put in a trivial wrapper on the client side in the following manner:

```
String      query(MDQuery query)
MDEntry[]   nextQuery(String token)
boolean     endOfQuery(String token)
```

The first `query` call would just return the token. The first call to the `nextQuery` call would return the list of attributes already in memory, retrieved by the actual server-side `query` call described above. The boolean method would just say whether the query is done. This would make sure that the user of the client does not make any mistake with the management of the token and the re-passing of the query object, resulting in unnecessary errors. The MDResult object would be managed completely by the client wrapper.

A sample client code iterating through a result would look like (in a language-neutral pseudocode)

```
String token = query(mdquery)

while ( !endOfQuery(token) ) {

   MDEntry[] results = nextQuery(token)
   // process results
}

endQuery(token)
```

2.2. SCHEMA OPERATIONS

This section defines the `MetadataSchema` interface. The Operations in this interface are the following:

`createSchema` Creates a new schema in the catalog.

`listEntrySchema` Lists all schemas associated with a given entry.

`addSchemaAttributes` Adds new attributes to an existing schema.

`removeSchemaAttributes` Removes attributes from an existing schema.

`renameSchemaAttributes` Renames an attribute in an existing schema.

`describeSchema` Get the full description of an existing schema in the catalog.

`dropSchema` Drops an existing schema from the catalog.

`listSchema` Lists all existing schemas in the catalog.

`addPolicy` Lists all existing schemas in the catalog.

`dropPolicy` Lists all existing schemas in the catalog.

2.2.1. CREATESCHEMA

```
void createSchema(String schemaName, Attribute[] attributes)
```

This method creates a new schema in the catalog. The new schema is associated with the list of attributes provided in the call. Each of this attributes will have its own name and type, given by the name and type fields in the `Attribute` object. The schema field is ignored. The value field MAY be used to store a default value for the attribute. See Section 2.6.1. for a description of the `Attribute` object. The attributes parameter is a list of `Attribute` objects.

Errors

`AuthorizationException` No access for client to create new schemas.

`ExistsException` A schema with the same name already exists in the catalog, or there is more than one attribute with the same name in the list of attributes given.

`InvalidArgumentException` One of the attributes given is invalid. It may be due to an invalid attribute name (i.e. empty), or the type given not being supported in the catalog.

`InternalException` Any other error on the server side (i.e. database down).

2.2.2. LISTENTRYSCHEMAS

```
String[] listEntrySchemas(String entry)
```

This method simply retrieves all schema names that are associated with a given entry.

Errors

AuthorizationException No access for client to the entry.

NotExistsException The entry does not exist.

InternalException Any other error on the server side (i.e. database down).

2.2.3. addSchemaAttributes

```
void addSchemaAttributes(String schemaName, Attribute[] attributes)
```

This method adds new attributes to an existing schema. Attributes MUST be unique within the schema.

The `schemaName` is the name of the schema where attributes should be added. The `attributes` parameter is simply a list of `Attribute` objects where the `name` and `type` field MUST be filled in.

Errors

AuthorizationException No access for client to add attributes to the schema.

NotExistsException The given schema does not exist in the catalog.

ExistsException There is already one attribute in the schema with the same name as one of the attributes in the list given. Or there is more than one attribute with the same name in the list.

InvalidArgumentException One of the attributes given is invalid. It may be due to an invalid attribute name (i.e. empty), or the type given is not supported in the catalog.

InternalException Any other error on the server side (i.e. database down).

2.2.4. removeSchemaAttributes

```
void removeSchemaAttributes(String schemaName, String[] attributeNames)
```

This method removes attributes from an existing schema. The semantics of the removal process are up to the implementation. It can remove the attribute from the schema even if there are entries in the catalog with values set for it. Or it can decide to remove only if there are no entries actually using this attribute at the time. It can also decide to remove an attribute from a schema when no entry in the catalog is associated with the schema at the time of the request.

The `schemaName` is the name of the schema where the attributes will be removed. The `attributeNames` is simply a list of the names of the attributes that should be removed from the schema.

Errors

AuthorizationException No access for removing attributes from the schema.

NotExistsException The given schema or one of the attributes given does not exist in the catalog.

InternalException Any other error on the server side (i.e. database down).

2.2.5. RENAMESCHEMAATTRIBUTES

```
void renameSchemaAttribute(String schemaName, String attributeName,
                           String newName)
```

This method renames an attribute in an existing schema. The `schemaName` is the name of the schema where the attribute will be renamed. The `attributeName` is the current name and the `newName` is the new name of the attribute.

Errors

`AuthorizationException` No access for changing attributes in the schema.

`NotExistsException` The given schema or one of the attributes given does not exist in the catalog.

`InternalException` Any other error on the server side (i.e. database down).

2.2.6. DELETESCHEMA

```
void deleteSchema(String schemaName)
```

This method removes an existing schema from the catalog. The semantics of this operation are up to the implementation. It may be that a schema can only be deleted if no entries in the catalog are associated with it at the time of the request. Or it may be that schemas are deleted even when there are entries associated with them. This would mean the existing metadata would be lost.

The `schemaName` is the name of the schema to be removed from the catalog.

Errors

`AuthorizationException` No access for client to delete the schema.

`NotExistsException` The given schema does not exist in the catalog.

`InternalException` Any other error on the server side (i.e. database down).

2.2.7. LISTSCHEMAS

```
String[] listSchemas()
```

This method lists all existing schemas in the catalog. The list of strings returned contains only the names of the schemas, not the whole description. To get the details on each of the schemas in the catalog, an additional request should be made to `describeSchema`, where all the attributes and their descriptions will be returned.

Errors

`AuthorizationException` No access to list schemas for the client.

`InternalException` Any other error on the server side (i.e. database down).

2.2.8. DESCRIBESCHEMA

```
Attribute[] describeSchema(String schemaName)
```

This method gets the full description of an existing schema in the catalog. A schema is described by the attributes it contains. Each of these attributes is represented as an `Attribute` object, so the description of a schema is simply a list of `Attribute` objects. The `schemaName` is the name of the schema to return the description of.

Return Value: A list of `Attribute` objects fully describing all elements of the schema. Each of these objects MUST have the `name` and `type` fields filled in. The `schema` field's may be empty. The returned array may be empty if the schema is empty.

Errors

`AuthorizationException` No access for client to get the schema description.

`NotExistsException` The schema requested does not exist in the catalog.

`InternalException` Any other error on the server side (i.e. database down).

2.2.9. ADDPOLICY

```
void addPolicy(String schemaName, String policy)
```

This method adds a new policy to the given schema. The policy is passed as a WHERE clause as defined in the MQL query language. The main example of use of policies is setting that only clients where the DN (Distinguished Name) taken from the certificate is equal to a given attribute are allowed to access the metadata of the items.

Errors

`AuthorizationException` No access for client to add policies to the schema.

`NotExistsException` The schema requested does not exist in the catalog.

`InvalidArgumentException` The policy passed as argument is invalid.

`InternalException` Any other error on the server side (i.e. database down).

```
void dropPolicy(String schemaName, String policy)
```

2.2.10. DROPPOLICY

This method removes an existing policy associated with the schema. There is no unique identifier to each policy. The whole string containing the existing policy is passed and matched against the policies associated with the schema. If there is a match, it is dropped.

Errors

`AuthorizationException` No access for client to drop policies from the schema.

`NotExistsException` The schema or the policy requested does not exist in the catalog.

`InternalException` Any other error on the server side (i.e. database down).

2.3. STANDALONE METADATA CATALOG OPERATIONS

This section defines the `MetadataCatalog` interface. The Operations in this interface are the following:

`createEntry` Creates new items/entries in the catalog.

`removeEntry` Removes items/entries from the catalog.

2.3.1. CREATEENTRY

```
void createEntry(MDEntry[] entries, String[] schemas)
```

This method creates new entries in the catalog. All entries given in the `entries` array will be associated to all the schemas in the `schemas` array. Additionaly the attribute values will be set according to the values in the `Attribute` array inside each `Entry` object. This array may be empty if no values should be set on creation. The `MDEntry` object is described in Section 2.6.2..

Errors

`AuthorizationException` No access for creating new entries.

`ExistsException` An entry already exists in the catalog.

`NotExistsException` A specified schema does not exist in the catalog.

`InvalidArgumentException` The given identifier for the new entry is invalid. Or some of the attributes given in the entry belong to a schema not listed in the `schemas` argument.

`InternalException` Any other error on the server side (i.e. database down).

```
void removeEntry(String query)
```

2.3.2. REMOVEENTRY

This method remove existing entries from the catalog. The `query` string should be a WHERE clause as defined in the 4. query language. It is a set of conditions that should result in a list of entry identifiers to be deleted.

Errors

`AuthorizationException` No access for deleting entries.

`InvalidArgumentException` The query string is invalid.

`InternalException` Any other error on the server side (i.e. database down).

2.4. PERMISSIONS

This section defines the access permissions for the `FASBase` interface. The Operations in this interface are the following:

`setPermission` Sets full set of permissions BasicPermission,ACL for a given item.

`getPermission` Retrieves all set permissions for given items.

`checkPermission` Checks if the current user has the required permission bits on the specified items.

2.4.1. SETPERMISSION

```
void setPermission(PermissionEntry[] permissions)
```

This method sets the full set of permissions BasicPermission,ACL for a given item, replacing any previous permissions. The `item` field in the `PermissionEntry` object may also contain a schema name. The `PermissionEntry` object is described in Section 2.6.5.

Errors

`AuthorizationException` No access right to update the permissions.

`NotExistsException` The item does not exist.

`InvalidArgumentException` Some part of the argument is invalid.

2.4.2. GETPERMISSION

This method retrieves all permissions for the given items. The `PermissionEntry` object is described in Section 2.6.5.

```
PermissionEntry[] getPermission(String[] items)
```

Errors

`AuthorizationException` No access right to get the permission information.

`NotExistsException` The item does not exist.

`InvalidArgumentException` Some part of the argument is invalid.

2.4.3. CHECKPERMISSION

```
void checkPermission(String[] items, Perm perm)
```

This method checks if the current user has the required permission bits given by the `Perm` object on the specified items and returns with an AuthorizationException if for some items this is not the case.

Errors

`AuthorizationException` The permission check failed - some entries cannot be accessed with the given Perm.

`NotExistsException` An item does not exist

2.5. SERVICE OPERATIONS

This section defines the `ServiceBase` interface. The Operations in this interface are the following:

`getVersion` Retrieve the implementation version.

`getInterfaceVersion` Retrieves the service interface version being implemented.

`getSchemaVersion` Retrieve the version of the schema being used.

`getServiceMetadata` Retrieve the value of a given key associated with this service.

2.5.1. GETVERSION

```
String getVersion()
```

Return the server implementation version as a string.

2.5.2. GETINTERFACEVERSION

Return the interface version as a string.

```
String getInterfaceVersion()
```

```
String getSchemaVersion()
```

2.5.3. GETSCHEMAVERSION

Return the schema version as a string.

2.5.4. GETSERVICEMETADATA

```
String getServiceMetadata(String key)
```

Service metadata query. The `key` parameter has to be specified. The method will return the requested parameter or an empty string if it does not exist.

2.6. AUXILIARY OBJECTS

In this section we define all the auxiliary objects that are being used in the interface methods described above.

2.6.1. ATTRIBUTE

```
Attribute {

    String schema
    String name
    String type
    String value

}
```

The `Attribute` object contains a schema, a name, a type and a value. It is the object with which items/entries in the Metadata Catalog can be associated.

An attribute in the Metadata Catalog is unique within a schema, in the sense that the combination name/type MUST be unique inside every given schema.

All values stored inside an Attribute are encoded as strings, even if the backend is storing them using a different type. The type of an Attribute is a hint on the content of this string, so that the backend can optimize storage and clients can decide how to present the information.

```
MDEntry {

    String entry
    Attribute[] attributes

}
```

2.6.2. MDENTRY

The `MDEntry` object contains the entry identifier and a list of attributes associated with the entry. It corresponds to an entry in the catalog, although it may contain only part of the attributes stored in the catalog.

2.6.3. MDQUERY

```
MDQuery {

    String query
    String type

}
```

A query on the metadata catalog is represented using the `MDQuery` object. The object contains the query itself, which is defined using the language specified in the `type` field.

An implementation of this interface may decide to support more than one query type. Even in this case, the basic query language defined as part of this interface, named MQL, MUST be supported by all the implementation to guarantee interoperability.

gLite intends to provide client-side utilities that convert BNL or XPath strings into MQL queries, which are very close to SQL queries.

2.6.4. MDRESULT

```
MDResult {

    Boolean done
    String token
    MDEntry[] entries

}
```

The `MDResult` object is returned by calls to `query` and `nextQuery`.

The `done` field tells the client if all results were retrieved, or if further `nextQuery` calls should be performed. The `token` field holds an identifier for the given request, which is useful for the client-side wrappers and stateful implementations.

Finally, the `entries` field holds the results of each iteration. The structure is that each item in the resultset is an entry or part of an entry in the catalog. If it happens that the only information to be retrieved is the identifier of the entry, then inside each MDEntry object the `attributes` field is null. Otherwise, the attributes are placed in this array, and associated with the entry they belong (allowing the retrieval of attributes for more than one entry in one call).

2.6.5. PERMISSIONENTRY

```
PermissionEntry {

    Permission permission
    String item
}

Permission {

    ACLEntry[] acl
}

ACLEntry {

    Perm principalPerm
    String principal
}

Perm {

    Boolean permission
    Boolean remove
    Boolean read
    Boolean write
    Boolean list
    Boolean execute
    Boolean getMetadata
    Boolean setMetadata
}
```

PermissionEntry contains a single item permission. The item is identified by a string, being the catalog entry item and the permission is a `Permission` object. The `Permission` object is composed of a list of `ACLEntry` objects, which in turn is a named principal (like a DN or a group name) and the associated permission bits, given by the `Perm` object. The single permission bits have different meanings for each of the metadata operations.

3. KNOWN ISSUES AND CAVEATS

Schema evolution is not being dealt with explicitly. The question how changes are being dealt with is left to the implementation. One of the suggestions is to have a timestamp-based rule of keeping track of

all changes. Another may be to assign both the new and the old schema to the entry, keeping track of the schema names somehow (but this would quickly lead to scaling problems). If an attribute is renamed, for example, how is this information to be kept using a time-stamp based approach?

Another issue is security. Current definition provides access control on the entry and schema level basis, and additionaly policies provide a way to restrict access to entries based on attribute values. The limitation of this model is that if access is granted or denied, the user will see all or none of the attributes inside the schema, respectively. There is no explicit way to define access control at the attribute level, and it is seen as too expensive to provide the same access control as done in entries and schemas - too fine-grained, and thus difficult to manage and performing poorly. A possibility is to have schemas defined in a way that this access restrictions are explicit, but it is not seen as a perfect solution either.

4. METADATA QUERY LANGUAGE (MQL)

Apart from having a common set of message exchanges between clients and service implementations, a common way for describing the queries being passed to the service is also necessary.

For this goal gLite has defined a query language very close to the SQL standard, but exposing only a subset of its functionality. The language is defined in a grammar using a BNF-like syntax, but specific to a tool called ANTLR. Converting the given syntax to be understood by another tool which follows BNF syntaxes should be straightforward.

4.1. QUERY EXAMPLES

MQL syntax has two main parts: SELECT and WHERE. The meaning is the same as in the SQL standard: SELECT defines the attributes that should be retrieved to the client, WHERE defines the condition that restricts the results.

The best way to introduce the language is by giving a few examples.

```
SELECT schemaOne.attr1ForSchemaOne, schemaTwo.attr1ForSchemaTwo
    WHERE schemaOne.attr2ForSchemaOne != null;
```

This is an extremely simple example. It states that both `attr1ForSchemaOne` from `schemaOne` and `attr1ForSchemaTwo` from `schemaTwo` should be retrieved to the client, but only for entries where `attr2ForSchemaOne` from `schemaOne` is not `null`.

In MQL all attributes MUST be namespaced (format being schemaName.attributeName). More complex queries can be made, involving functions and operators, or condition clauses including the operators `LIKE` and `IN`. The following example includes some of this functionality.

```
SELECT (schemaOne.attr1ForSchemaOne + schemaTwo.attr1ForSchemaTwo),
    sum(schemaTwo.attr2ForSchemaTwo) + schemaThree.attr1ForSchemaThree,
    WHERE schemaOne.attr2ForSchemaOne LIKE '%sample%'
    AND schemaThree.attr2ForSchemaThree = schemaOne.attr1ForSchemaOne;
```

Apart from using the user-defined schemas, one can also use the pre-defined schemas describe in section 1.2.. As an example of the usage of this virtual schema information, imagine a doctor wanting to retrieve all information from all his patients. A possible query would be:

```
SELECT patientInfo.* WHERE patientInfo.doctor = request.clientDN;
```

4.2. LANGUAGE GRAMMAR

Several internal ANTLR definitions were removed to make the grammar more clear. The definition of the rules and tokens are not affected.

```
//////////////////////
// MQL Parser Rules //
//////////////////////

query
: "select" select_list "where" condition (SEMI)?
;

select_list
: column ( (COMMA) column )*
;

column
: ( schema_name DOT ASTERISK ) => schema_name DOT ASTERISK
| ( expression ("as" identifier)* ) => expression ("as" identifier)*
;

expression
: term ( ( PLUS | MINUS ) term )*
;

term
: sub_term ( ( ASTERISK | DIVIDE ) sub_term )*
;

sub_term
: sql_literal
| function OPEN_PAR column ( COMMA column )* CLOSE_PAR
| OPEN_PAR expression CLOSE_PAR
| schema_name DOT attribute_name
;

function
: numeric_function
| char_function
| group_function
;

numeric_function
: "abs" | "cos" | "log" | "mod" | "pow" | "rnd" | "sin" | "sqrt"
;
```

```
char_function
: "concat" | "lower" | "length" | "substr" | "upper"
;

group_function
: "avg" | "count" | "min" | "max" | "sum"
;

sql_literal
: NUMBER | QUOTED_STRING | "null"
;

attribute_name
: identifier
;

schema_name
: identifier
;

comparison_op
: EQ | LT | GT | NOT_EQ | LTE | GTE
;

identifier
: IDENTIFIER
;

condition
: condition_term ( OR condition_term )*
;

condition_term
: condition_subterm ( AND condition_subterm )*
;

condition_subterm
: ( expression comparison_op expression )
=> expression comparison_op expression
| ( OPEN_PAR condition CLOSE_PAR ) => OPEN_PAR condition CLOSE_PAR
| ( expression "like" sql_literal ) => expression "like" sql_literal
| ( expression "in" sql_literal_list )
=> expression "in" sql_literal_list
;

sql_literal_list
: OPEN_PAR sql_literal ( COMMA sql_literal )* CLOSE_PAR
;

////////////////////////////////
```

```
// MQL Parser Token Definition //
////////////////////////////////

IDENTIFIER
: 'a'..'z'('a'..'z'|'0'..'9'|'_'|'$'|'#')*
;

QUOTED_STRING
: '\'' ( ~'\'' )* '\''
;

NUMBER
: '0'..'9' ('0'..'9')* ( (DOT) ('0'..'9') )*
;

EQ
: '='
;

NOT_EQ
: "<>" | "!=" | "^="
;

LT
: '<'
;

GT
: '>'
;

LTE
: "<="
;

GTE
: ">="
;

OPEN_PAR
: '('
;

CLOSE_PAR
: ')'
;

PLUS
: '+'
;
```

```
MINUS
: '-'
;

ASTERISK
: '*'
;

AND
: "and" | "&&"
;

OR
: "or" | "||"
;

DOT
: '.'
;

COMMA
: ','
;

SEMI
: ';'
;

WS
: (' ' | '\t' | '\r' '\n' | '\n')
;
```

REFERENCES

[1] N. Santos B. Koblitz. A Proposal for a Metadata Interface. Technical Note, January 2005. http://agenda.cern.ch/askArchive.php?base=agenda&categ=a05664&id=a05664s1t1/document.

[2] F. Carminati, P. Cerello, C. Grandi, E. Van Herwijnen, O. Smirnova, and J. Templon. Common Use Cases for a HEP Common Application Layer – HEPCAL. Technical report, LHC Computing Grid Project, 2002. http://project-lcg-gag.web.cern.ch/project-lcg-gag/LCG_GAG_Docs/HEPCAL-prime.pdf.

[3] F. Carminati and J. Templon (Editors). Common Use Cases for a HEP Common Application Layer for Analysis – HEPCAL II. http://lcg.web.cern.ch/LCG/SC2/GAG/HEPCAL-II.doc.

[4] JRA1 Data Management Cluster. *Catalog User Guide*. EGEE, March 2005. https://edms.cern.ch/document/570780.

[5] JRA1 Data Management Cluster. *Overview of gLite Data Management*. EGEE, March 2005. https://edms.cern.ch/document/570643.

[6] Steven Hanlon et. al. Unlucky for Some: The thirteen core use cases for HEP metadata. Technical Note, December 2004. http://www.gridpp.ac.uk/datamanagement/metadata/SubGroups/UseCases/CoreUseCases_v10.pdf.

[7] EGEE Project Technical Forum Requirements Database. https://savannah.cern.ch/support/?group=egeeptf.