Open Service Interface Definition

Authorization

Document Release: 2.0
OSID 2.0

Summary

The OSIDs cover services that are likely to involve restricted content and operations. The Authorization OSID provides a way to define who is authorized to do what, when.

Authorizations associate Agents, which represent the user or another actor in the system, with Functions and Qualifiers. Before performing an operation, implementations will want to know if there is an Authorization that covers the operation. If the Agent is authorized, the operation can be attempted. If not, the operation is not allowed.

Service Definition

One can think of the Authorization system in terms of a grammar where an Agent is a noun, Functions are operations or verbs and Qualifiers are objects of the operation. An Authorization can then be read as a sentence. For example: Jeff (an Agent) can write checks (a Function) on the Academic Computing (a Qualifer) account. Since a system may have many Qualifiers and those Qualifiers may be well represented using a hierarchy, there is support for such a hierarchy in the OSID. For example: the Academic Computing Account (a Qualifier) can have sub-accounts for Software Development (another Qualifier) and Educational Technology Consulting (a third Qualifier). If Jeff (our Agent) is explicitly authorized to write checks (the Function) on the Academic Computing Account (the explicitly stated Qualifier), he is implicitly authorized to write checks on the elements of the Academic Computing Account (i.e. the Software Development account and the Educational Technology Consulting account.

Functions and Qualifiers can be further organized by Type. For example: writing checks (a Function) might be an administrative function (a FunctionType) or an academic function (another FunctionType). The Academic Computing (a Qualifier) might be a Department (a QualifierType) or a Center (another QualifierType).

An Agent in the authorization OSID is represented by a unique identifier (agentId). This means that, if an application using the authorization OSID wants to get more information about a particular Authorization's Agent, it must go to an implementation of the Agent OSID or some other external mechanism to convert the agentId into detail information about the Agent.

The semantic meaning of AgentIds, Functions, FunctionTypes, Qualifiers, and QualifierTypes is the responsibility of the application using the authorization service. The authorization service is intended to persist these Functions, FunctionTypes, Qualifiers, QualifierTypes, Qualifier hierarchies, and Authorizations. An Authorization is the persisted combination of an AgentId, a Function, and a Qualifier. An application can as the authorization service for all the information associated with Authorizations in a variety of ways.

The Authorization OSID is intended to support several different kinds of applications and implementations:

- Allows checking if an Agent or the User is Authorized to perform some action. No management or display of Authorizations would be supported.
- Allows both checking Authorizations and management and display of Authorizations, but not reorganizing the Qualifier Hierarchy. That task might be performed using a Hierarchy OSID implementation and some other application.
- Allows checking, Authorization managing, and Qualifier Hierarchy management.

Definition of Terms

Authorization – the association of an Agent, a Function, and a Qualifier. Authorizations may have effective and expiration dates or may always be active.

AgentId– a unique identifier representing a principal or actor known to a system.

Function – an action or role that can be authorized. A Function has one and only one FunctionType.

FunctionType – a characterization for Functions.

Qualifier – an identifier for the object of a Function's action. A Qualifier has one and only one QualifierType.

QualifierType – a characterization for Qualifiers.

User – the current user of the system.

Qualifier Hierarchy – an organization of Qualifiers into nodes such that all authorizations granted for an ancestor node are granted to all descendant nodes.

Explicit Authorization – an Authorization created using the AuthorizationManager or otherwise that associates an Agent, Function, and Qualifier. Explicit Authorizations can be modified.

Implicit Authorization – an Authorization that was not created using the AuthorizationManager and has the Agent, Function, and Qualifier or Qualifier Descendant (see Qualifier Hierarchy) of an Explicit Authorization. Implicit Authorizations, unlike Explicit ones, cannot be modified directly. They can be modified only through an Explicit Authorization containing an ancestor Qualifier.

org.osid.authorization.AuthorizationManager

An examination of an Open Service Interface Definition (OSID) begins with the Manager. All Managers[1] provide the way to create the objects that implement the principal interfaces in the Service.

The AuthorizationManager has methods which server three purposes:

1. setting up Authorizations
2. answering which Authorizations, Functions, and Qualifier exist and who is authorized to do what
3. answering what an Agent or the user is Authorized to do

(1) Authorization Setup

Authorizations, which can be in effect either for a specific period of time or always, are the union of three elements: an Agent, a Function, and a Qualifier.

[1] org.osid.OsidManager defines the interface extended by the Managers in each OSID. Here we will be discussing org.osid.authorization.AuthorizationManager. Refer to OsidManager for more information.

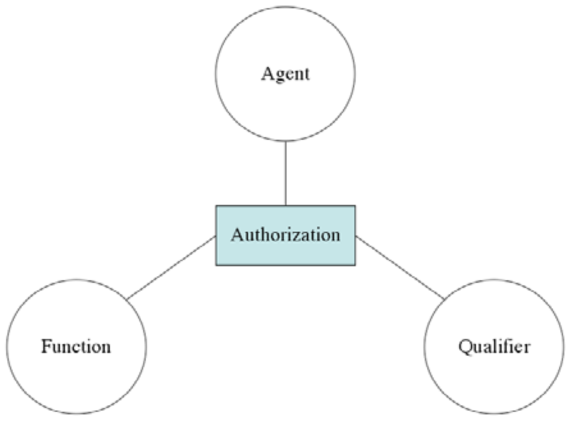

Figure 1: Elements in an Authorization

AgentIds represent agents known to the system. The Authentication OSID implementation is responsible for identifying the user. The Agent OSID implementation is responsible for detail information associated with the AgentID. The agent referenced in an Authorization may or may not be the current user.

Functions represent activities and operations that Agents can perform. Each Function has a Function-Type. For example, there might be Functions Types for creating, adding, editing, viewing or getting, and removing things. There might be a Function for creating Canonical Courses and a Function for creating Course Offerings. Both might have the same create Function Type. There is nothing in the Authorization OSID that specifies what Functions are defined – that is up to the implementation.

Qualifiers represent what is acted upon by a Function. Qualifiers have Qualifier Types. Qualifier Types exist to constrain the range of valid Qualifiers. For example, if there is a Function for editing Course descriptions (this might have an Editing Function Type) the Qualifier might be a specific Course Offering id and the Qualifier Type might be Course Offering. To reduce the number of explicit Authorizations needed, Qualifiers can be arranged hierarchically. This makes it possible that Authorizations for a specific agent do not need to exist for every Qualifier. Such a requirement of one Authorization per Qualifier could vastly increase the number of Authorizations. Any Agent for whom there is an Authorization for a given Qualifier is assumed also to have the Authorization for any Qualifier that is a descendent of the explicit Qualifier.

For example, assume we have a hierarchy of Qualifiers representing Courses in an English Department. If and Agent has the authority to make description changes with a department-level Qualifier, this would imply that the same Agent is Authorized to change the description for any Course whose Qualifier is a child of the department-level Qualifier.

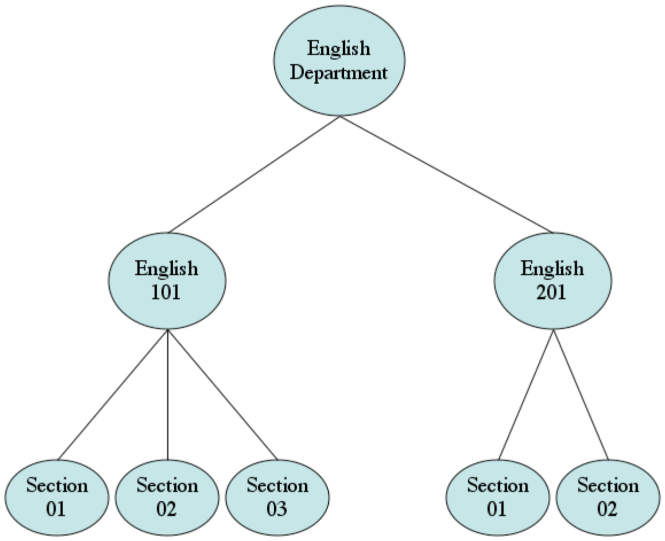

Figure 2: Sample Hierarchy of Qualifiers

The AuthorizationManager includes a variety of methods for set-up. Some implementations will expose the methods for Authorization management, specifically creating and deleting Authorizations, Functions, and Qualifiers. It may also be the case that these entities are already in placed or managed through another process. More simply, an application may not focus on creating and deleting these objects.

AuthorizationManager Method Summary

Returned Object	Method
Authorization	createAuthorization(Id agentId, Id functionId, Id qualifierId)
Authorization	createDatedAuthorization(Id agentId, Id functionId, Id qualifierId, long effectiveDate, long expirationDate)
Function	createFunction(Id functionId, String referenceName, String description, Type functionType, Id qualifierHierarchyId)
Qualifier	createQualifier(Id qualifierId, String referenceName, String description, Type qualifierType, Id parentId)
Qualifier	createRootQualifier(Id qualifierId, String displayName, description, Type qualifierType, Id qualifierHierarchyId)
void	deleteAuthorization(Id authorization)
void	deleteFunction(Id functionId)
void	deleteQualifier(Id qualifierId)
boolean	supportsDesign()
boolean	supportsMaintenance()

(2) Querying Existing Authorizations, Functions, and Qualifiers

The AuthorizationManager has methods to answer questions relating to what Authorizations, Functions, and Qualifiers exist and about who can do what, when. There are different methods for getting subsets of all the objects available. If one is seeking information about Authorizations, one can ask:

- Which Authorizations exist for a specific combination of Function and Qualifier? These are further filtered based on whether the Authorization is or is not effective.
- Which Authorizations exist for a combination of any Function of a specific Function Type and a Qualifier?
- Which Authorizations exist for a specific agent and the above combinations?
- Which Authorizations exist for a specific agent and a specific Qualifier or any descendant of that Qualifier? The assumption is that Authorizations are explicitly created for a given Agent at the highest appropriate Qualifier node in the Qualifier hierarchy.

As an example, assume the following has been set up:

Function Types

Create
Edit

Qualifier Types

Course Offering
Course Section

Functions

Create Course Offering
Create Course Section
Edit Course Offering
Edit Course Section

Qualifiers

English 101
English 101 Section 01
English 101 Section 02
English 101 Section 03
English 201
English 201 Section 01
English 201 Section 02
English 201 Section 03

Explicit Authorizations

Agent	Function	Qualifier
Professor A	Create Course Offering	English 101
Professor A	Edit Course Offering	English 101
Teaching Assistant 1	Edit Course Offering	English 101 Section 01
Teaching Assistant 2	Edit Course Offering	English 101 Section 02
Teaching Assistant 2	Edit Course Offering	English 101 Section 03
Professor B	Create Course Offering	English 201
Professor B	Edit Course Offering	English 201
Teaching Assistant 3	Edit Course Offering	English 201

If I ask which Authorizations exist for the Function Edit Course Offering and the Qualifier English 101 Section 01, the implementation should return:

Agent	Function	Qualifier
Professor A	Edit Course Offering	English 101
Teaching Assistant 1	Edit Course Offering	English 101 Section 01

Professor A does not have an explicit Authorization for English Section 01, but since English 101 is a ancestor in the Qualifier hierarchy, it is assumed Professor A has the same Authorization for Section 01.

If I ask which Agents can Edit Course Offering for English 101 Section 01, the implementation should return:

Agent
Professor A
Teaching Assistant 1

The Qualifier hierarchy may have multiple roots. A Qualifier's children are one level beneath in the hierarchy. A Qualifier's descendants include the children and all their children until no children remain.

There is an **agentExists**() method that tests if the Authorization implementation recognizes a specific Agent by Id. This test is whether the Authorization implementation knows this Id, distinct from asking an agent OSID AgentManager implementation a similar question. This is no requirement that the authorization OSID implementation and agent OSID implementation rely on the same list of Agents.

There are many variations on a question regarding available Authorizations. One can be interested in Authorizations for the user, an Agent, or anyone. One can be interested in Authorizations for a specific Function (by Id) or more widely for an entire FunctionType. One can be interested in Explicit Authorizations, which can be modified, or for the potentially much larger number of Implicit Authorizations, which can be modified only indirectly through an Explicit Qualifier's entry. Here is a table of the kinds of questions and the method used to get the answer. Note that, as with any operation, what information is returned is subject to the user's Authorization to receive the information.

Getting Authorizations is one line of inquiry. Another is to ask which Agents are authorized to perform specific Functions or Types of Function for certain specific Qualifiers or branches of a Qualifier hierarchy.

Here are all the methods relating to these queries:

AuthorizationIterator	getAllAZs(Id agentId, Id functionId, Id qualifierId, boolean isActiveNowOnly)
AuthorizationIterator	getAllAZsByFuncType(Id agentId, Type functionType, Id qualifierId, boolean isActiveNowOnly)
AuthorizationIterator	getAllUserAZs(Id functionId, Id qualifierId, boolean isActiveNowOnly)
AuthorizationIterator	getAllUserAZsByFuncType(Type functionType, Id qualifierId, boolean isActiveNowOnly)
AuthorizationIterator	getExplicitAZs(Id agentId, Id functionId, Id qualifierId, boolean isActiveNowOnly)
AuthorizationIterator	getExplicitAZsByFuncType(Id agentId, Type functionType, Id qualifierId, boolean isActiveNow)
AuthorizationIterator	getExplicitUserAZs(Id functionId, Id qualifierId, boolean isActiveNowOnly)
AuthorizationIterator	getExplicitUserAZsByFuncType(Type functionType, Id qualifierId, boolean isActiveNowOnly)
Function	getFunction(Id id)

FunctionIterator	getFunctions(Type functionType)
TypeIterator	getFunctionTypes()
Qualifier	getQualifier(Id qualifieId)
QualifierIterator	getQualifierChildren(Id qualifierId)
QualifierIterator	getQualifierDescendents(Id qualifierId)
IdIterator	getQualifierHierarchies()
TypeIterator	getQualifierTypes()
QualifierIterator	getRootQualifiers(Id qualifierHierarchyId)
AgentIterator	getWhoCanDo(Id functionId, Id qualifierId)

(3) What is an Agent or the User Authorized to do

To use most if not all services the OSIDs include would require some Authorization. In some cases there is a one-to-one mapping between an OSID method and an authorizable operation. In other cases, several OSID methods together comprise a single authorizable operation. It is likely that implementations will often call the AuthorizationManager to see if the user is Authorized to perform a specific Function with a specific Qualifier. The method they will call is **isUserAuthorized**(). *Note that it is not intended that implementations should use other Manager methods for this purpose.*

A similar method, **isAuthorized**() answers the same question for an Agent other than the user. Both **isAuthorized**() and **isUserAuthorized**() test for effective Authorizations only. This is the equivalent of using an *isActiveNowOnly* argument set to true in other methods.

If the method returns true, the operation should proceed. If the method returns false, the implementation will likely throw the OSID-specific Exception and use the defined PERMISSION_DENIED message.

org.osid.authorization.Authorization

The Authorization interface connects an AgentId, a Function, and a Qualifier. These are all passed into the **createFunction**() method in the Manager. Authorizations have an effective date and an expiration date. These optionally be passed into the **createFunction**(). The effective and expiration dates can be updated. The authorization OSID implementation updates modifiedBy and determines isActiveNow and isExplicit.

Authorization Method Summary

long	getEffectiveDate()
long	getExpirationDate()
Id	getModifiedBy()
long	getModifiedDate()
Function	getFunction()
Qualifier	getQualifier()
Id	getAgentId()

boolean	isActiveNow()
boolean	isExplicit()
void	updateExpirationDate()
void	updateEffectiveDate()

org.osid.authorization.Function

The Function interface simply covers attributes of the Function. Functions have an immutable reference name, a description, a FunctionType, a QualifiedHierarchyId, and an Id[2]. These are all passed into the **createFunction**() method in the Manager. The description can be updated.

Function Method Summary

String	getDescription()
String	getReferenceName()
Type	getFunctionType()
Id	getId()
Id	getQualifierHierarchyId()
Void	updateDescription(String description)

org.osid.authorization.Qualifier

The Qualifier interface covers the attributes of the Qualifier as well as methods for manage and interrogate the Qualifier hierarchy. Qualifiers have a display name, a description, a QualifiedType, and an Id. These are all passed into the **createQualifier**() or **createRootQualifer**() methods in the Manager. The display name and description can be changed while the other attributes can not.

The Qualifier hierarchy is composed and maintained using **addParent**(), **changeParent**(), and **removeParent**() methods[3]. The Qualifier hierarchy is interrogated using **isParent**(), **getChildren**(), **isChildOf**(), and **isDescendantOf**(). To get all the descendants, call **getChildren**() recursively.

[Insert a discussion of qualifier hierarchy composition, modification, and sample results of interrogations]

Qualifier Method Summary

void	addParent(Id parentQualifierId)
void	changeParent(Id oldParentId, Id newParentId)
QualifierIterator	getChildren()
String	getDescription()
String	getReferenceName()

[2] Only in this OSID and Hierarchy are Ids passed in rather than assigned by the implementation.

[3] Certain modifications to a hierarchy would result in an inconsistent state and an Exception will be thrown.

Id	getId()
QualifierIterator	getParents()
Type	getQualifierType()
boolean	isChildOf(Id parentId)
boolean	isDescendentOf(Id ancestorId)
boolean	isParent()
void	removeParent(Id parentQualifierId)
void	updateDescription(String description)

org.osid.authorization.*Object* Iterators

Iterators return a set of elements of a specific type, one at a time. The purpose of all Iterators is to offer a way for OSID methods to return multiple values of a common type and not use an array. Returning an array may not be appropriate if the number of values returned is large or is fetched remotely. Iterators do not allow access to values by index, rather you must access values in sequence. Similarly, there is no way to go backwards through the sequence unless you place the values in a data structure, such as an array, that allows for access by index.

All iterators contain two methods. The **hasNext<*Object type*>**() method returns true if there are more values of the iterator type available; false otherwise. The **next<*Object type*>** () method returns the next element in the sequence. Note that in many cases, the order of elements is not guaranteed. The following iterators are included in this OSID:

AuthorizationIterator

FunctionIterator

QualificationIterator

Type iterators, such as those used for Function and Qualifier Types are defined in org.osid.shared.TypeIterator. Similarly, the AgentIterator is defined there.

org.osid.authorization.AuthorizationException

The OSIDs make use of Exceptions as a mechanism for responding to error or unusual conditions. All methods in the authorization OSID throw an **AuthorizationException**. The Exception contains a message that is a String. The following message Strings are defined in AuthorizationException:

- EFFECTIVE_PRECEDE_EXPIRATION
- CANNOT_DELETE_LAST_ROOT_QUALIFIER

If an implementation uses these messages, consumers of the implementation can easily test and conditionally respond to the Exception. Note that other kinds of Exception constructors are not used as all do or can devolve to a String. All methods of all interfaces of all OSIDs throw a subclass of org.osid.OsidException. This requires the caller of any implementation method handle the Exception.

If a method performs an operation without incident, an object or primitive may be returned, but in most cases, methods do not return error codes or a success or failure boolean. For example, a method that deletes an object with a particular identifier, would throw an Exception if the identifier were unknown; the method would not return, for example, false.