Author:	
Last Modified:	2005-01-22T21:32:00Z
By:	

Table of Contents

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

Introduction

The Authorization API describes the mechanisms used by the Service Broker to assign different levels of access to Service Broker functionality to different users. The Service Broker attempts to provide good, common sense default behavior that can be tailored to a particular institution or class's needs using the Service Broker's administrative web interface. This document describes the low level internal calls that implement this business logic. This document, therefore, addresses developers who will maintain and extend the Service Broker rather than administrators who will configure a particular Service Broker installation. The introductory material in the first half of this document, however, will provide useful background for administrators, but they will not need to master the actual method calls discussed in the second half.

The approach taken here was inspired by the Authorization API of the Open Knowledge Initiative (OKI: http://web.mit.edu/oki/). The OKI Authorization and associated APIs had not stabilized at the point the Service Broker implementation had to go ahead. In a future Service Broker release, we may bring the Service Broker Authorization API more strictly in accord with OKI.

Grants

Service Broker permissions are implemented using two mechanisms. Experiment records and other user-generated documents preserve the identity of their creator ("the owner"). "The owner" generally retains special permissions over such documents. Other permissions implementing class and site policy are established using an administrative interface that manages instances of the **`Grant`** type. A

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

Grant instance combines three concepts: the *agent* (either a single user or a group of users gaining the permission specified by the **Grant**), the *function* (which specifies the type of permission), and the *qualifier* (which specifies the Service Broker resource to which the new permission applies). The **Agent** is specified by the **int** ID of the appropriate **User** or **Group**. The **function** is drawn from a fixed set of **strings** known to the Service Broker implementation, e.g., *"useLabServer"* or *"administerGroup"*. The anatomy of the **Qualifier** will be discussed in detail below, but a simple example will help to get us started. If the **function** of a **Grant** is the **String** *"useLabServer"*, then we would expect the **Qualifier** to specify which Lab Server the **Grant** applies to.

```
public struct Grant
{
     public int    grantID
     public int    agentID
     public string function
     public int    qualifierID
}
```

Figure

The following table lists the **function strings** defined to implement the authorization policy for the first generation Service Broker. Type checking between **function** and **Qualifier** type is not enforced, so the second column only indicates typical usage. This allows a

Lab Server Web Service API

Service Broker

**Lab
Server**

Trusted Code To Check Authorization

**Client
Application**

Browser

particular Service Broker implementation to extend the set of known **Qualifier** types.

Function	Normal Qualifier Type
useLabClient	**LabClient**
useLabServer	**LabServer**
readExperiment	**Experiment** or **ExperimentCollection**
writeExperiment	**Experiment** or **ExperimentCollection**
addMember	**Group**
administerGroup	**Agent**
superUser	

The **superUser function** (which has no context) implies all other grants as well as being required for the most privileged administrative functions, such as registering a new Lab Server.

Qualifiers

Agents form an implicit hierarchy by virtue of the structure of group membership. Members of **Groups** can be **Users** or other **Groups**. **Qualifiers** form an explicit hierarchy; that is, administrators and Service Broker default logic explicitly construct the **Qualifier** hierarchy to model nested permission domains appropriate to site policy.

An example will make this clearer (please refer to on page 6). At MIT, 1.00 is the designation of one of the basic programming courses. If this course were to use a Service Broker to administer online lab sessions, a standard way to organize students and staff would be to

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

make the students members of a **Group** with the **groupName**, “1.00”, and course staff members of a **Group** with **groupName** “1.00Staff”. As we shall see, by making the 1.00Staff group a subgroup of group 1.00, staff members will inherit all the privileges of students in the class. Typical administrative policy would grant course staff control over all class resources, e.g., student experiment records. Individual students have permission to view and modify their own experiment records because they “own” those resources. By default, they may neither view nor modify experiments that belong to anyone else. The professor teaching 1.00 might want to allow all students to view experiment results from all students but only to modify or delete their own.

How would we establish this particular policy using **Grants** and **Qualifiers**? There are two separate sets of permissions to be established: (1) students can view all experiment records but only write their own, and (2) course staff can read and write all experiment records. The Service Broker automatically creates a **Qualifier** for each experiment record when the experiment is run. It is default policy to create these **Qualifiers** as the children of a single parent **Qualifier** of type *“ExperimentCollection”*. To implement the first set of permissions, an administrator would create a **Grant** that would associate the 1.00 group with the “readExperiment” **function** on the *“ExperimentCollection”* **Qualifier** node belonging to the class. This would allow members of the 1.00 group to read all resources that are descended from the *”ExperimentCollection”* node. Staff would gain the same access because the staff group is a subgroup of the 1.00 group. To implement the second set of permissions, the administrator would need to create a second **Grant** associating the more restrictive staff subgroup with the more powerful *“writeExperiment”* **function** applied to

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

the same *“ExperimentCollection”* **Qualifier** node. Now let’s look at these mechanisms in more detail.

Qualifiers possess a **qualifierReferenceID** (**int**), **qualifierTypeID** (**int**), and **qualifierName** (**string**) specified at the time the **Qualifier** is created via the **AddQualifier()**method. This method assigns the new **Qualifier** a unique **int qualifierID**. The **qualifierReferenceID** acts, in database terms, like a foreign key and identifies a resource represented by the **Qualifier**. It will usually be an ID in the ID space corresponding to the **qualifierTypeID**. We stated above that the Service Broker creates a **Qualifier** whenever a new experiment record is stored. The **qualifierReferenceID** of this **Qualifier** is the same as the experiment ID, and the **qualifierTypeID** corresponds to “Experiment”.

Some **Qualifiers** do not possess a **qualifierReferenceID** (==-1). This usually occurs when administrators want to group **Qualifiers** of a given type into a collection represented by a new **Qualifier**, in order to simplify the assignment of permissions, as in the case of the *“ExperimentCollection”* node above. As an additional example, if students in a particular class will be executing experiments on three separate Lab Servers, it might be useful to represent each Lab Server by a **Qualifier** of type “LabServer”, and then to make these **Qualifiers** the children of a new **Qualifier** that represents the set of three Lab Servers. The **qualifierReferenceID**s of the first three **Qualifiers** will be the corresponding LabServer IDs. But what should be the **qualifierReferenceID** of the fourth **Qualifier**, the one that represents the collection? Since the collection of Lab Servers is solely defined by the hierarchy of **Qualifiers** and is not represented by any

Service Broker

Lab Server

Trusted Code To Check Authorization

Client Application

Browser

other object in the system, there is no reasonable candidate for the **qualifierReferenceID**; it is set to the special value -1.

In order to administer **Grants**, however, we must be able to unambiguously identify this collection node. We might want to add a new Lab Server to the collection by linking the **Qualifier** representing the new Lab Server in as a child. More specifically, a Qualifier must possess a **qualifierName** if it does not have a **qualifierReferenceID** (==-1). If the **qualifierReferenceID** is not -1, then the combination of **qualifierTypeID** and **qualifierReferenceID** must be unique. If **qualifierReferenceID** is -1, then the combination of **qualifierTypeID** and **qualifierName** must be similarly unique. This implies that a resource can only appear once in the **Qualifier** hierarchy.

Qualifier is the class that implements this concept internally.

```
public struct Qualifier
{
     public int     qualifierID;
     public int     qualifierReferenceID;
     public int     qualifierType;
     public string  qualifierName;
     public Qualifier[] parents;
}
```

Figure

Explicit vs. Implicit Grants

Service Broker

Lab Server

Trusted Code To Check Authorization

Client Application

Browser

A call to **AddGrant()** (see page 12) defines an explicit **Grant** or permission. The Service Broker authorization architecture also considers *implicit* grants when checking authorization via **CheckAuthorization()** and other calls. An implicit grant is a grant implied by an existing explicit grant and the **Agent** and **Qualifier** hierarchies that have been created.

More formally, a **Grant {agentID1, function, qualifierID1}** will be considered valid if the grant is explicitly registered with the Service Broker or if there is a second grant **{agentID2, function, qualifierID2}** explicitly registered for which

1. **agentID1=agentID2** or **agentID1** is a descendant of **agentID2**, and
2. **qualifierID1=qualifierID2** or **qualifierID1** is a descendant of **qualifierID2**.

If it is not the case that **agentID1 = agentID2** and **qualifierID1 = qualifierID2**, then the second **Grant** is only implied by the first. The second **Grant** does not possess its own **grantID** or its own **Grant** object.

As an example, consider the two hypothetical **Group***s*, 1.00 and 1.00Staff, used in the example above. Students gain permission to read other students' experiment records and course staff to delete those records through implicit grants that are implied by the corresponding explicit grants on the "ExperimentCollection" **Qualifier**.

Let us work through a more complicated example (illustrated in on page 8). The **Group** objects representing both these groups possess a required unique **string groupName** and an optional **string**

Lab Server Web Service API

Service Broker

Lab Server

Trusted Code To Check Authorization

Client Application

Browser

description. In this example, the **string**s "**1.00**" and "**1.00Staff**" serve as the **groupName**s for their respective groups. A user wishing to invoke the **ListMemberIDsInGroup()** method on the **1.00** group requires an *"administerGroup"* **Grant** for the group. The **groupID** corresponding to group 1.00 cannot serve as a **Qualifier** directly, but the system can create a **Qualifier** that represents group **1.00**. This **Qualifier** will have an arbitrary, unique **qualifierID** (*qualID1*) but its **qualifierTypeID** will correspond to **Group** and its **qualifierReferenceID** will be that of group **1.00**, i.e. the **groupID** for the **Group** that the **Qualifier** represents. A second **Qualifier** with the **qualifierReferenceID** corresponding to 1.00Staff (*qualID2*) will represent the group containing the course's teaching staff.

The head teaching assistant for the course (**userName=jsmith**) will be a member of the **1.00Staff** group, and may well possess the following explicit **Grant**: **{***grantID1***, userID** of **"jsmith", "administerGroup",** *qualID2***}**. This grant would allow him to list the IDs of all members in the **1.00Staff** group (and add new members to the group). But note that this **Grant** doesn't give **jsmith** any **Grant**, explicit or implied to administer the supergroup 1.00. If **jsmith** also possesses the **"administerGroup" Grant** on the **Qualifier** with the reference to **Group 1.00**, he may choose to add the **Grant {***grantID2***, groupID** of **"1.00Staff", "administerGroup",** *qualID1***}.** The two most important points to note from this example are, first, that a **Grant** identifies the target of its function indirectly through the **qualifierReferenceID** of its **Qualifier**, and second, that although subgroups inherit their parent group's **Agent** permissions, **Qualifiers** of subgroups do not inherit

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

permissions on their supergroups. If they did, then all staff members could add new members to the **`1.00Staff`** group.

Figure

Service Broker Authorization Policy and Architecture

Authorization requires both an internal system architecture and an external social design. The social design or policy may differ from campus to campus. The Service Broker should not specify and enforce a particular set of rules that stipulate who can do what. It should allow system administrators to implement whatever policy a particular campus requires.

At the same time, the Service Broker architecture and implementation must follow certain internal practices if security and authorization policy is to be consistently enforced. The user interacts with the Service Broker through a web browser or a client application. The low level Service broker APIs (*Service Broker Administrative API, Service Broker Experiment Storage API, Service Broker Authorization API, Service Broker Authentication API)* assume they are executed by trusted code that checks the user's permission to perform the actions in question by calling methods in the *Service Broker Authorization (Authz) API* discussed in this document. The user's identity must have been determined previously using mechanisms defined in the *Service Broker Authentication (Authn) API*.

Service Broker

Lab Server

Trusted Code To Check Authorization

Client Application

Browser

Service Broker Default Authorization Policy

Service Broker authorization policy should be table driven so that it can be configured for individual campuses. The default policy that will be implemented by the distribution version of the Service Broker without any customization is expressed in the following table. Each row of the table corresponds to one of the methods available in the insecure low level APIs (Authorization, Authentication, Administrative, Experiment Storage) or the top level *Client To Service Broker API*. If the *Function* is listed as *trusted*, then the method should only be called during the execution of trusted administrative code and not directly at the behest of a user. The *anyone* designation indicates that any user may execute the method. The *owner* designation indicates that only the owner of the designated item(s) may execute the method. In all other cases, the row corresponding to a method lists the *Function* and *QualifierType* of the default **Grant** required to execute the method at the command of a user (**Agent**).

We expect that the mapping of **Grant** to API method will be implemented using tables loaded at Service Broker startup. This implies that the authorization policy could be modified at a particular site by modifying these configuration files.

Authorization API

Method	*Function*	*QualifierType*
AddGrant	*superUser*	
RemoveGrants	*superUser*	
ListGrantIDs	*anyone*	
GetGrants	*anyone*	

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

FindGrants	*anyone*	
CheckAuthorization	*trusted*	
AddQualifier	*trusted*	
AddQualifierParent	*trusted*	
GetQualifierID	*trusted*	
GetQualifier	*trusted*	
RemoveQualifierParent	*trusted*	
SetQualifierParent	*trusted*	
RemoveQualifiers	*trusted*	
ListQualifierIDs	*trusted*	
ListQualifierParents	*trusted*	
ListQualifierChildren	*trusted*	
ListQualifierDescendants	*trusted*	
IsQualifierDescendant	*trusted*	

Authentication API

Method	*Function*	*QualifierType*
Authenticate	*trusted*	
CreateNativePrincipal	*superUser*	
RemoveNativePrincipals	*superUser*	
ListNativePrincipals	*superUser*	
SetNativePassword	*superUser*	

Administrative API

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

Method	***Function***	***QualifierType***
AddLabServer	*superUser*	
RemoveLabServers	*superUser*	
ModifyLabServer	*superUser*	
ListLabServerIDs	*anyone*	
GetLabServers	*anyone*	
GenerateServerID	*superUser*	
GetServerID	*superUser*	
GenerateIncomingServerPasskey	*superUser*	
GetIncomingServerPasskey	*superUser*	
RegisterOutgoingServerPasskey	*superUser*	
GetOutgoingServerPasskey	*superUser*	
AddLabClient	*superUser*	
RemoveLabClients	*superUser*	
ModifyLabClient	*superUser*	
ListLabClientIDs	*anyone*	
GetLabClients	*anyone*	
SetUserXMLExtensionSchema	*superUser*	
AddUser	*addMember*	*Group*
RemoveUsers	*superUser*	
ModifyUser	*superUser*	
ListUserIDs	*superUser*	
ListOrphanedUserIDs	*superUser*	
GetUsers	*administerGroup*	*Group*
GetUsersNames	*anyone*	

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

AddGroup	*superUser*	
RemoveGroups	*superUser*	
ListGroupIDs	*anyone*	
GetGroups	*superUser*	
AddMemberToGroup	*addMember*	*Group*
RemoveMembersFromGroup	*administerGroup*	*Group*
ListUserIDsInGroupRecursively	*administerGroup*	*Group*
ListMemberIDsInGroup	*administerGroup*	*Group*
ListGroupsForAgent	*anyone*	
ListGroupsForAgentRecursively	*anyone*	
IsAgentMember	*anyone*	
NotifyUsers	*administerGroup*	*Group*
SaveClientItemValue	*owner*	
GetClientItemValue	*owner*	
RemoveClientItems	*owner*	
ListClientItems	*owner*	
AddSystemMessage	*administerGroup*	*Group*
RemoveSystemMessages	*administerGroup*	*Group*
ModifySystemMessage	*administerGroup*	*Group*

Experiment Storage API

Method	***Function***	***QualifierType***
SetResultXMLExtensionSchema	*superUser*	
SetBlobXMLExtensionSchema	*superUser*	
SaveExperimentSpecification	*writeExperiment*	*Experiment*

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

RetrieveExperimentSpecification	*readExperiment*	*Experiment*
SaveSubmissionReport	*writeExperiment*	*Experiment*
SaveResultReport	*writeExperiment*	*Experiment*
RetrieveExperimentResult	*readExperiment*	*Experiment*
RetrieveLabConfiguration	*readExperiment*	*Experiment*
SaveExperimentAnnotation	*writeExperiment*	*Experiment*
RetrieveExperimentAnnotation	*readExperiment*	*Experiment*
RemoveExperiments	*writeExperiment*	*Experiment*
GetExperimentInformation	*readExperiment*	*Experiment*
FindExperimentIDs	*readExperiment*	*Experiment*
SaveBlobObject	*writeExperiment*	*Experiment*
GetBlobObjects	*readExperiment*	*Experiment*
RemoveBlobObjects(int[])	*writeExperiment*	*Experiment*
RemoveBlobObjects(int[],int[])	*writeExperiment*	*Experiment*
GetBlobInformation	*readExperiment*	*Experiment*

Client to Service Broker API

Method	***Function***	***QualifierType***
GetLabStatus	*useLabServer*	
GetEffectiveQueueLength	*useLabServer*	
GetLabInfo	*useLabServer*	
GetLabConfiguration	*useLabServer*	
Validate	*useLabServer*	
Submit	*useLabServer*	
Cancel	*useLabServer*	

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

GetExperimentStatus	*useLabServer*	
RetrieveResult	*useLabServer*	
SaveClientItem	*owner*	
GetClientItem	*owner*	
RemoveClientItem	*owner*	
ListClientItems	*owner*	
Annotate	*writeExperiment*	*Experiment*

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

Methods

Grant Management Methods

AddGrant

Purpose:

/* Adds a grant to the agent (User or Group) agentID. */

Arguments:

`int agentID`
/* The ID of the agent to which the grant is to be added. */

`string function`
/* The function of grant to be added; function must be one of the fixed set of **`Function`** strings recognized by the Service Broker. */

`int qualifierID`
/* The ID of a previously created **`Qualifier`** whose **`qualifierType`** is appropriate to the function, if the **`Function`** type requires one; otherwise null. */

Returns:

`int ID`
/* The **`ID`** of the new Grant if

- **`agentID`** specifies a registered user or group
- **`function`** is a legitimate **`function`** type

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser

- **qualifierID** is a recognized ID of an appropriate **qualifierType** for the function (in the case where **qualifierID** is specified).

In addition, the specified grant must not already have been made to the specified agent. If any of these conditions fails, then this method returns null. Note that most of the other Service Broker **AddXXX()** methods specify the ID of the object being added; the **AddGrant()** method does not do this; **grantID**s are arbitrary **int**s generated by the Service Broker implementation. */

RemoveGrants

Purpose:

/* Removes previously registered grants. The removal of a grant should not affect any Service Broker calls currently in progress, but it can prevent the execution of later related method calls. For instance, if the revoked grant permits a group to use a particular lab server, then any experiments in progress should run to completion, but users will be unable to retrieve the corresponding results through the Service Broker. */

Arguments:

int[] grantIDs
/* An array of the IDs of previously registered grants. */

Returns:

int[] IDs[]

Lab Server Web Service API

Service Broker

Lab
Server

Trusted Code To Check Authorization

Client
Application

Browser