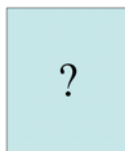




Open Service Interface Definition

Authentication



Document Release: 2.0
OSID 2.0

Summary

The Authentication Open Service Interface Definition (OSID) primarily supports invoking an authentication process. The implementation of this Service is responsible for gathering whatever information is appropriate to perform the authentication. The Service also supports testing if a user is authenticated, returning the Agent Id that corresponds to the user, returning the types of authentication available, and destroying either all authentications or only those for a given type.

The OSIDs can interact with information and resources over which some form of access control is required. Authentication and Authorization work together where Authentication ensures interactions are at the request an identified user and Authentication reports what the identified user can do.

Service Definition

An examination of an Open Service Interface Definition (OSID) usually begins with the Manager. All Managers¹ provide the way to create the objects that implement the principal interfaces in the Service. Before discussing the Manager in detail, we will review the intended role of the Authentication Service overall.

The focus of the Authentication OSID is authenticating users and mapping a user to the AgentId representing them in the system. The OSID does not cover services for maintaining authentication data such as password list, accounts etc.

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

A central theme with O.K.I. is that services are defined in order to structure the way in which applications interact with back-end system. For many of these back-end systems, users may only perform certain authorized operations, for example, getting particular content or changing particular data. The Authorization service concerns itself with answering who is permitted to do what, when. Some applications will want to Authorize many operations, perhaps in fine detail, while others will use Authorization more lightly. In order to use an Authorization, systems need to know that the user is who they say they are. Authentication is the service that fulfills this need.

The Authentication service is designed to allow for support of more than one Authentication process. For example, the same Authentication implementation could support authenticating users through user id and password, certificates, or any other method. An Authentication Type is included in many of the AuthenticationManager's methods. A Type is provided when asking to authenticate the user and when destroying Authentications. Each implementation returns the set of Authentication Types it supports.

org.osid.authentication.AuthenticationManager

The **authenticateUser()** method accepts a single argument: an Authentication Type and invokes a process to authenticate the user. A Type is four Strings that taken together characterize something, in this case the kind of authentication². Note that we do not give the method a user representation, such as an Agent, and ask that Agent to be authenticated. We ask if the user is authenticated using an particular Authentication Type and the implementation of the **authenticateUser()** method is responsible for invoking whatever process is need to do this. Such a process might present a user id and password dialog and test the input; or to gather a user's certificate; or some other process. While only one user per instance of the AuthenticationManager³ is assumed, a user of the authentication service may want to re-authenticate the current user. **authenticateUser()** should be implemented to support this scenario as well. For such a case, the implementation may always return true for systems that do not re-test authentication during a session or it may perform any appropriate test and return true or false.

The Authentication service can be implemented to use any of a wide range of commonly used authentication systems⁴ as well as more individual systems. The integration details are left to the implementation and are not spelled out in the OSID.

The **isUserAuthenticated()** method also accepts an Authentication Type and returns a boolean (true or false) that indicates whether the user is or is not authenticated.

The **getUserId()** method performs a mapping between the user and an Agent in the system for a specific AuthenticationType. Agents represent users and are input arguments to certain methods in other OSIDs. The method returns the Agent's unique identifier⁵. To convert this id into an Agent, one can use the **getAgent()** method in a Agent OSID's implementation of AgentManager.

AuthenticationManager Method Summary

void	authenticateUser(Type authenticationType)
void	destroyAuthentication()

² A more detailed discussion of Types in general is documented elsewhere.

³ A system wishing to allow a change of user could log out the current user and authenticated a new user, but the model is only one user per instance of the AuthenticationManager.

⁴ There are many such systems. An example is Kerberos, a network authentication protocol designed to provide strong authentication for client/server applications by using secret-key cryptography. Another is SAML.

⁵ There is a detailed discussion of Ids in separate documentation.

void	destroyAuthenticationForType(Type authenticationType)
TypeIterator	getAuthenticationTypes()
Id	getUserId(Type authenticationType)
boolean	isUserAuthenticated(Type authenticationType)

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 SID 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.

AuthenticationTypes are returned using TypeIterator defined in the Shared OSID.

org.osid.authentication.AuthenticationException

The OSIDs make use of Exceptions as a mechanism for responding to error or unusual conditions. All methods in the authentication OSID throw a **AuthenticationException**. The Exception contains a message that is a String. Messages for AuthenticationExceptions are defined in SharedException and OsidException.

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.