Common Security Framework

Author: Zlatko Michailov
Version: 0.3 (30-Mar-2005)

Authentication

The purpose of the authentication mechanism is to verify whether the caller is who he/she claims he/she is. In case that is confirmed, a tangible SessionContext is produced. The exact authentication method is only relevant to the final SessionContext as much as the SessionContext may be required to maintain a proprietary session token.

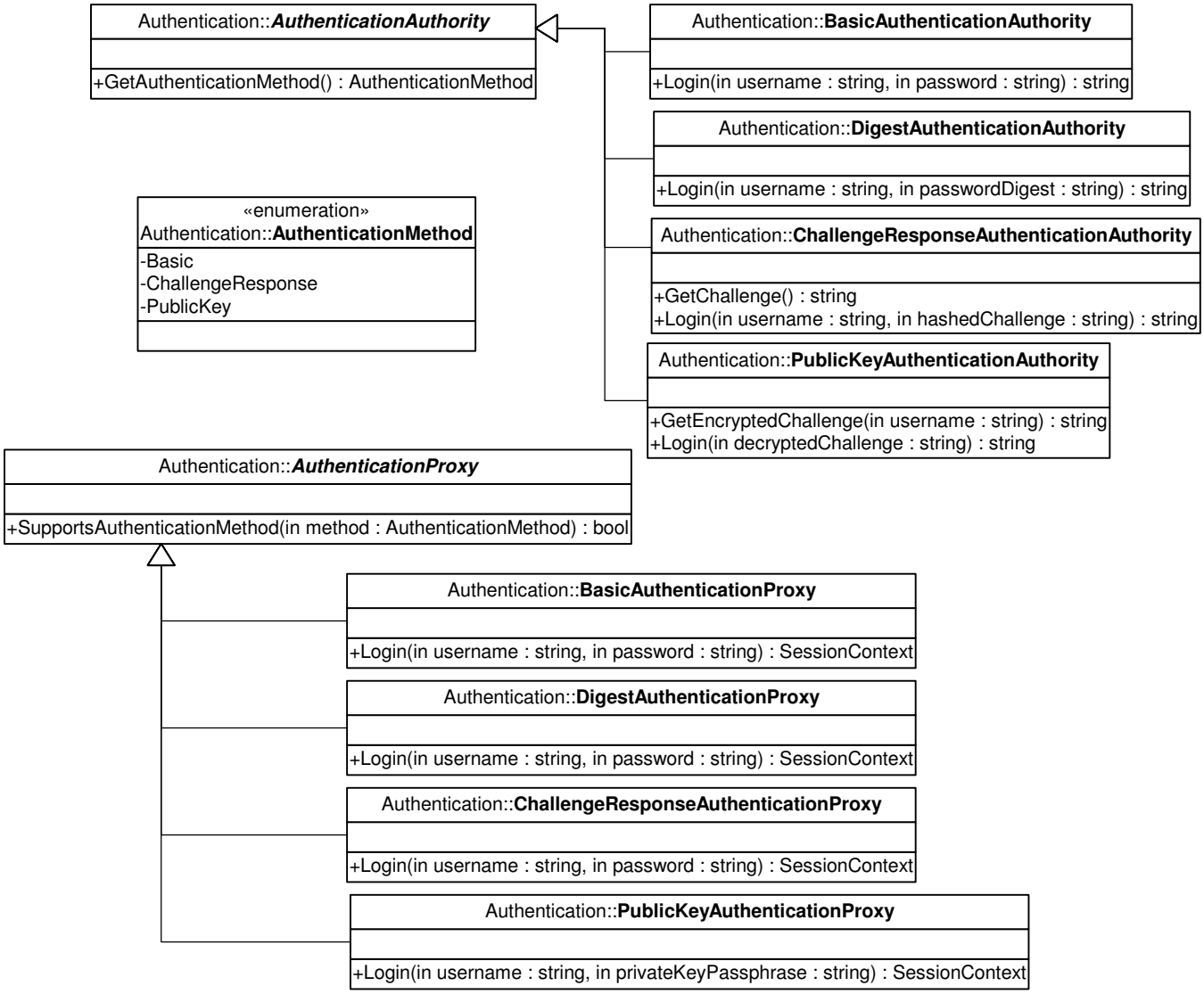
The authentication process has two sides - Authentication Authority and Authentication Proxy. In case the credentials match, the AUTHENTICATION AUTHORITY responds with a proprietary session token that the Authentication Proxy should store in the SessionContext. The AUTHENTICATION PROXY is responsible for communicating with the Authentication Authority and for producing a SessionContext in case the Authority responds positively.

BASIC AUTHENTICATION
The password must be sent to the authority. Therefore, in order for this authentication method to be secure, communication must be encrypted by the transportation layer.

DIGEST AUTHENTICATION
A digest of the password must be sent to the authority. Although the password itself is not sent, this method is still unsecure because the password digest is static. Once it is intercepted, an impersonator may use it until the user changes their password. Communication must be encrypted as for Basic Authentication.

CHALLENGE-RESPONSE AUTHENTICATION
The proxy obtains a random number (challenge) from the authority. Then the proxy hashes that challenge with the password and responds back with the hashed challenge.

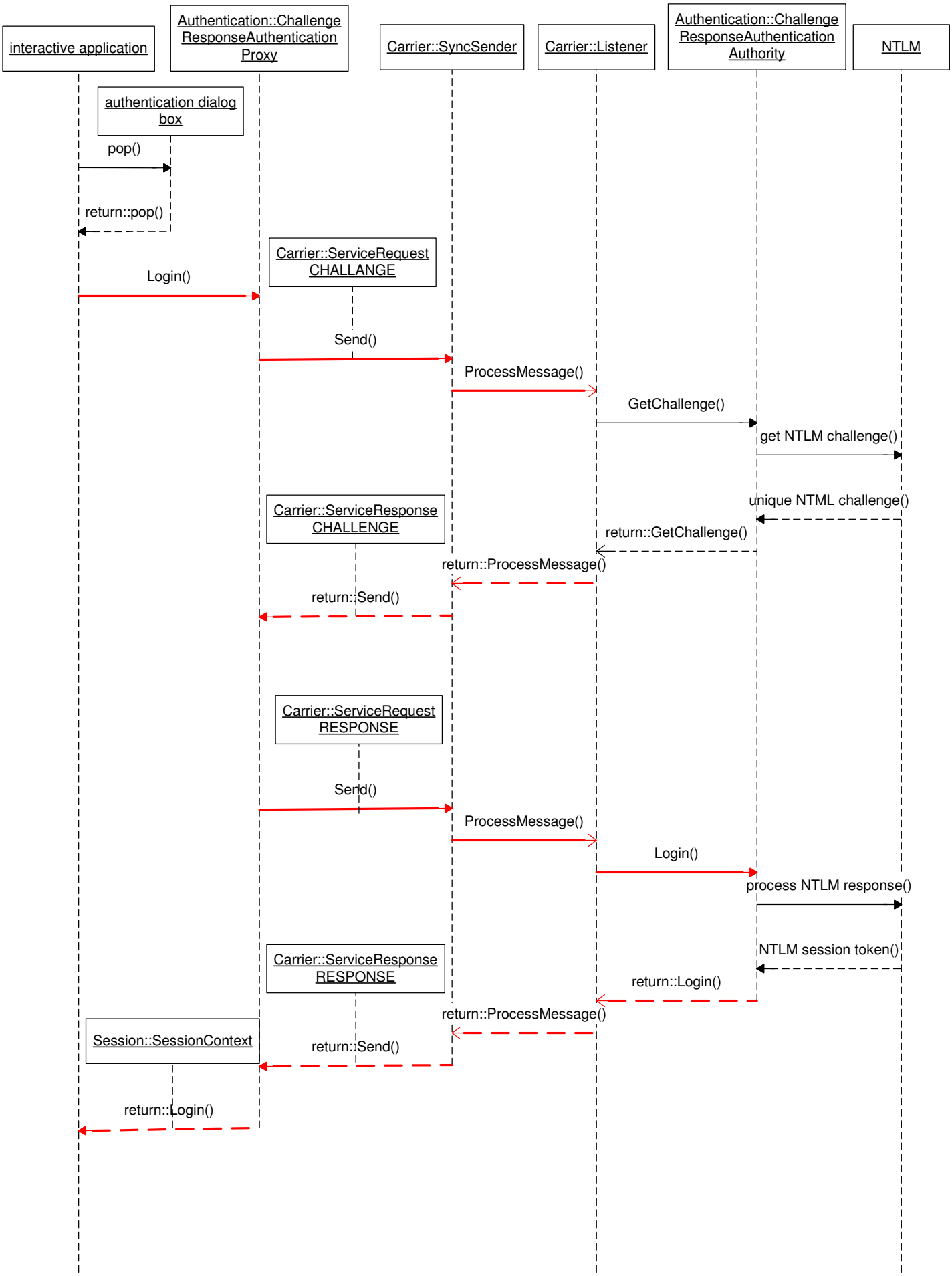
PUBLIC KEY AUTHENTICATION
The public key of the user must be stored at the authority beforehand. The proxy obtains a random number (challenge) encrypted with the public key of the user from the authority. Then the proxy decrypts that challenge with the private key of the user and responds back with the decrypted challenge.



Authentication Samples

```
// INTERACTIVE CLIENT APPLICATION
// ------------------------------------------------------------------------------


Authentication.AuthenticationMethod method;
string                              username;
string                              pass;
Session.SessionContext              sessCtx;


// Allow the interactive user to choose the authentication method
app.privatePopAuthenticationDialogBox( out method, out username, out pass );


// Act according to the chosen authentication method
switch ( method )
{
    case Authentication.AuthenticationMethod.Basic:
        // pass is the real password.
        // It will be sent to the authentication provider.
        // Optionally warn the user abot it.

        basicProxy = new Authentication.BasicAuthenticationProxy();
        sessCtx = basicProxy.Login( username, pass );
        basicProxy = null;
        break;


    case Authentication.AuthenticationMethod.ChallangeResponse:
        // pass is the real password but
        // it will not be sent to the auth. provider.

        crProxy = new Authentication.ChallangeResponseAuthenticationProxy();
        sessCtx = crProxy.Login( username, pass );
        crProxy = null;
        break;


    case Authentication.AuthenticationMethod.PublicKey:
        // pass is the passphrase for the private key file.
        // The proxy will find the private key file and will use pass to read it.

        pkProxy = new Authentication.PublicKeyAuthenticationProxy();
        sessCtx = pkProxy.Login( username, pass );
        pkProxy = null;
        break;
}


// At this point sessCtx is initialized.



// Application code
// ...



// Logout
sessCtx.Logout();
```

Authentication Sample Sequence Diagram - Challenge-Response



```
// BASIC AUTHENTICATION AUTHORITY
// ------------------------------------------------------------------------------


public string Login( string username, string password )
{
    PrivateSessionToken token = null;

    if ( privateVerifyPassword( username, password ) )
    {
        token = privateCreateSession( username );
    }

    return token.ToString();
}
```

```
// CHALLENGE-RESPONSE AUTHENTICATION AUTHORITY
// ------------------------------------------------------------------------------


public string GetChallenge()
{
    return privateGenerateUniqueChallenge();
}


public string Login( string username, string hashedChallenge )
{
    PrivateSessionToken token = null;

    if ( privateVerifyChallengeResponse( hashedChallenge, username ) )
    {
        token = privateCreateSession( username );
    }

    return token.ToString();
}
```

```
// PUBLIC KEY AUTHENTICATION AUTHORITY
// ------------------------------------------------------------------------------


public string GetEncryptedChallenge( string username )
{
    string challenge     = privateGenerateUniqueChallenge();
    string userPublicKey = privateLoadUserPublicKey( username );

    return privateEncryptChallenge( challenge, userPublicKey );
}


public string Login( string decryptedChallenge )
{
     PrivateSessionToken token    = null;
     string              username = null;

    if ( privateVerifyChallengeResponse( decryptedChallenge, out username ) )
    {
        token = privateCreateSession( username );
    }

    return token.ToString();
}
```

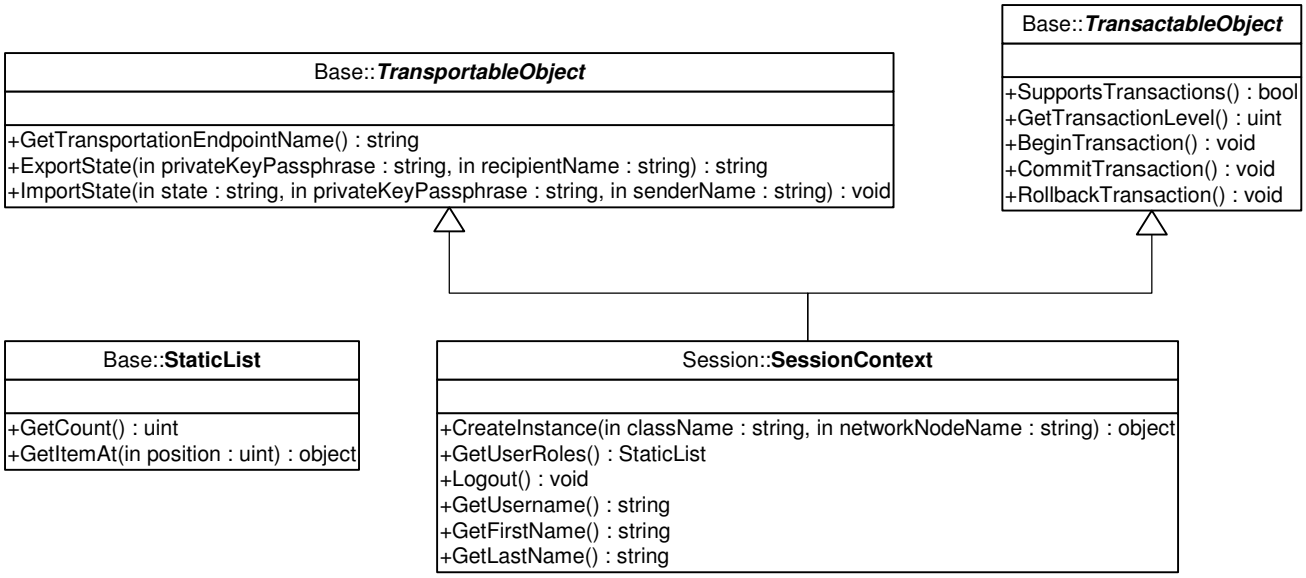
Session

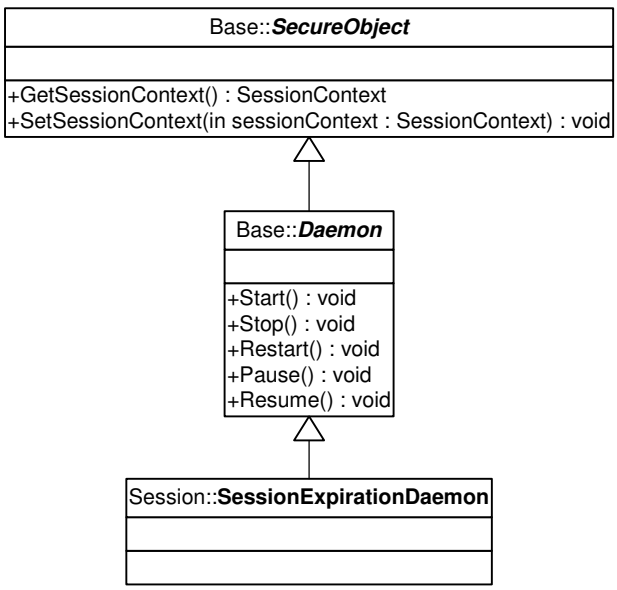
A SECURE OBJECT can obtain a reference to its session context at any time to verify the permissions of the session. A secure object may also make its own session context propagate over other secure object instances. Not all objects are secure. For instance static pieces of data may not require any access restriction. However, objects that provide access to data stores should be secure.

A TRANSPORTABLE OBJECT can serialize its current state into a text string so that it could be rebuilt at a remote location without any loss of data. This may require serializing of any inherited, embedded and/or referenced object instance as well. Thus, in order for any secure object to be transportable, SessionContext must be transportable. To prevent intruders from intercepting serialization packets, the serialized content must be encrypted so that only the receiver may decrypt it. It is not necessary to use asymmetric encryption as long as the sender and the receiver have exchanged a private symmetric key. Furthermore, to prevent intruders from tempering with the packets, the serialization content must be signed with the private key of the sender.

A TRANSPORTATION ENDPOINT may be a network node, or a process on a network node that has its own pair of a public and a private key. The endpoint name is used to identify the key pair to use for encryption/decryption during transportation.

Designing TRANSACTABLE OBJECTS is a difficult task in the general case. However, when objects represent data entities from a single database, it is possible to use database transactions to make the objects transactable. How to bind multiple objects into a transaction? Since objects are bound by a session context, if the session context is transactable, a transaction will cover all the objects of that session. Constraint: once a transaction is started, a database connection must be exclusively locked in the database connection pool until the transaction is committed/rolled back. This disables the transportability of the session context temporarily because database connections are not transportable.





Access Control

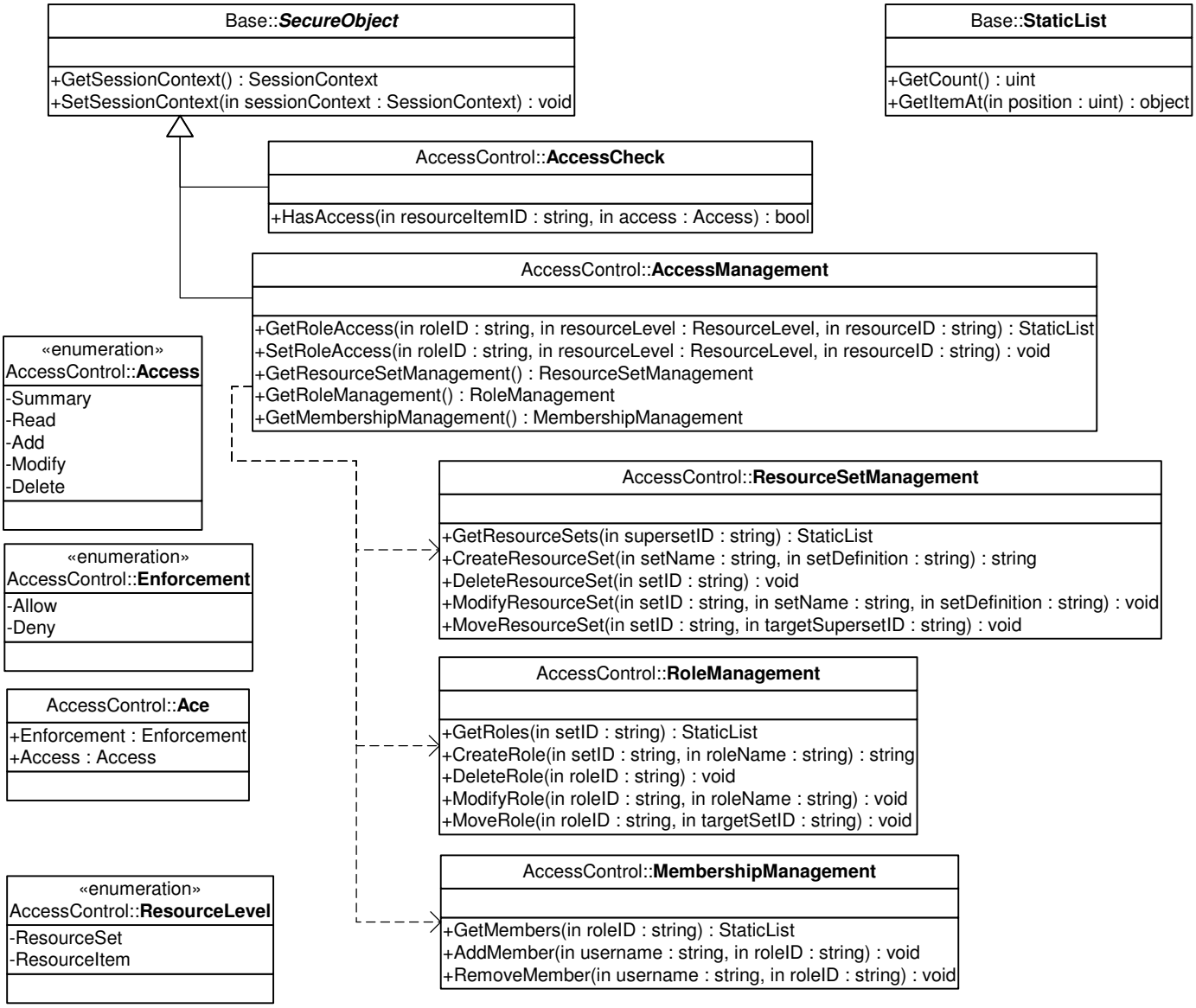
ITEM-LEVEL ACCESS CONTROL means that access to each item is controlled individually. This approach is suitable for low-level systems like OS where there are few generic sets: Files, Folders, and Processes, and items of the same type may belong to completely different workflows. The main advantage is granularity. The main drawbacks are scalability and maintenance - the number of ACEs is a multiple of the total number of items in the system.

SET-LEVEL ACCESS CONTROL means that access is controlled for all the items of a given set as one single entity. This approach is suitable for high-level application systems where there is a variety of properly-defined resource sets, and where all the items of a certain set fall in well-known workflows. Examples of sets are: entity types, branches of a hierarchy, object classes, class methods. The main advantage is related to adding, moving, and removing items - since an item only inherits its set's ACEs, there is no need to make any ACE adjustments when the above operations are performed. On top of that, it offers better maintenance and scalability than item-level access control because the number of ACEs is of the magnitude of the number of sets which should be limited and relatively static. The main drawback is granularity.

GROUP-BASED- and ROLE-BASED ACCESS CONTROL are two aspects of the same thing. While groups represent the physical aggregation of people, roles represent the purpose of the aggregation. For instance, the users who act as administrators belong to group Administators, and the guests to the system belong to group Guests. Neither of those two aspects touches the most imprtant issue - the level of granularity - item or set. For clarity this security system uses the term ROLE to unite a users into a set.

THIS SECURITY SYSTEM employs a mixed model where both item- and set-level access control are supported. Furthermore, sets may be nested, and roles may be attached to a set at any nested level or not attached to a set at all. In the latter case the role's permissions would apply to all items in the system.

Each secure object's method must verify the calling session's access every time it is invoked. There is a single access control repository, access to which is split into Access Check and Access Management. ACCESS CHECK allows any session to check whether it has a given permission over a given resource set/item.ACCESS MANAGEMENT modifies the repository and is only available to sessions with proper permissions.



Access Control Samples

```
// ACCESS CHECK CLIENT
// ------------------------------------------------------------------------------


// This is a method of a secure object that returns items that
// match the given criteria and the calling user is allowed to read.

public Base.StaticList findSecureDataItems( string criteria )
{
    ListBuilder restricted = null;

    // Obtain a list of all items that match the criteria
    Base.StaticList all = privateQueryAll( criteria );

    Session.SessionContext sessCtx = GetSessionContext();
    AccessControl.AccessCheck accessCheck = sessCtx.CreateInstance(
            "AccessControl.AccessCheck", null );

    // For each item in the list check access.
    // In order for this to be efficient, a good caching mechanism must be used
    // by the implementation of AccessControl.AccessCheck.HasAccess().

    for ( uint i = 0; i < all.GetCount(); i++ )
    {
        PrivateItem item = all.GetItemAt( i );
        string resourceID = item.getID();
        if ( accessCheck.HasAccess( resourceID, AccessControl.Access.Read ) )
        {
            restricted.add( item );
        }
    }

    return restricted.ToStaticList();
}
```

```
// ACCESS CHECK AUTHORITY
// ------------------------------------------------------------------------------


public bool HasAccess( string itemID, AccessControl.Access access )
{
    Session.SessionContext sessCtx   = GetSessionContext();
    Base.StaticList        userRoles = sessCtx.GetUserRoles();
    Base.StaticList        sets      = null;
    bool                   hasAccess = false;

    do
    {
        bool hasDirectAccess;

        // checkDirectAccess() checks direct access to the specified
        // item through the specified list of resource sets. Then
        // it replaces the list of sets with a list of sets of the next
        // nested level to which the item belongs. The process
        // stops when the new list of (nested) sets is empty.

        hasDirectAccess = checkDirectAccess( itemID, userRoles, inout sets );
        if ( sets.GetCount > 0 )
        {
            hasAccess = hasDirectAccess;
        }
    }
    while ( sets.GetCount() > 0 )

    return hasAccess;
}
```

Carrier

A CARRIER mechanism is responsible for delivering messages between network nodes. For instance, it is used to implement transportable objects. Also, it is used to connect an authentication proxy with an authentication authority. A carrier could be either synchronous or asynchronous.

A MESSAGE is the envelope that wraps the actual payload to be carried over. A message itself is not a transportable object. The payload object must be. And, the payload object must make sure its serialized state is properly encrypted. For convenience in security implementation, specific service request classes may be defined as transportable objects.