Authentication/Authorization/Identification SDK

Component History

Version	Component
V 1.0	Java and C++ web-based game client (deprecated)
V 1.5	Java, C++ and Shockwave web-based game client
V. 2.0	Added standalone game client login and game server authorization

Document History

Document name and Version	Author	Reviewed By	Release Date
Authentication/Authorization/Identification SDK V 1.0	Steve Keller and Kent Holman		10/00

FAQ

What platforms does Auth/Auth/Ident support?

Windows 95, 98,and NT for C, C++ development environments, Java for cross platform development environments and Solaris and Linux for game server libraries.

1. What are the system requirements for Auth/Auth/Ident?

   Refer to the EA.com Platform Requirements document.

2. What do I get with the Auth/Auth/Ident SDK?

   - A readme text file that lists the files included in the SDK (For example documents and code).
   - A Programming Guide that includes the C, C++, and Java APIs, and usage examples for C, C++ and Java.

3. What tools do I need to develop with the Auth/Auth/Ident SDK?

   - Visual C++ v6.0 and Service Pack 3.

4. What components do I need that are not included in the SDK ?

   You need to have Wininet.

5. What about security?

   - The browser-game component uses an opaque, randomly generated “ticket” so that no user specific information is available.

- The standalone login component uses https (hyper text transfer protocols) to communicate with EA.com so all data is encrypted.

- Communication from game servers to EA.com is on secure back channel that is not visible to the public Internet.

Introduction

The Auth/Auth/Ident system provides authentication, authorization, and identification services to game clients and to other Game Technology or Enterprise Group systems via a specific API (Application Programming Interface). The system does not provide initial login or authentication for players on EA.com; it provides additional authentication and authorization services for players who are already logged in to the Web site.

Game clients use this service - certain games (e.g., subscription games) must undergo an additional authorization step to prove that a player is allowed to play a particular game at a particular time

The Auth/Auth/Ident SDK (Software Development Kit) includes the following sections:

Auth/Auth/Ident Browser-based Games Client Usage Instructions, the Authorization Client APIs for Browser-based games, Auth/Auth/Ident Standalone Games Client Usage Instructions, Embedded Game Client APIs for Standalone Games, the AuthAuthIdent Game Server Library API and several code usage examples of C, C++, and Java implementations for both browser-based and standalone game clients.

Features

- Authorization of browser-based game clients
- Secure login of standalone game clients
- Backend authorization of game servers (for use with standalone game clients).

Auth/Auth/Ident Browser-based Games Client Usage Instructions

This section provides information on integrating the EA.com Games Technology Authorization service into browser-based client programs, such as online games or client matchmaking components. These components provide access to the Authorization service, while hiding the complexity of making a connection to the service.

Java Implementation

The Java implementation of the Authorization service for browser-based game clients consists of the following classes:

Client Classes

Authorizer
AuthorizeException
Ticket

All of the previous classes reside in the package **com.ea.authorize**. You must include these classes in a Java game to use the Authorize service.

Supporting Classes

AuthorizerDefaultImpl
TicketDefaultImpl
NVParser

Clients should never access any of the supporting classes directly; access to the service is through the interface classes only. This allows EA.com to modify the implementation of the client classes without affecting client code.

The one exception to this is the actual construction of the class that implements the Authorizer interface, AuthorizerDefaultImpl. Rather than burden this lightweight component with a factory class, a decision was made to provide access to the Authorizer implementation directly. If the name of this class (or the signature of its constructor) changes in the future, there will be a minor, one-line change to client code.

Clients using the Authorizer interface need to obtain three pieces of information from the execution environment: a **Context id** (or "**ticket"**), a **product id** and a **service URL** for the Authorization service. In Java applets, these are typically passed in via the **<PARAM>** HTML tag inside the **<APPLET>** tag.
The client retrieves an instance of a "ticket" by calling Authorizer.authorizeTicket(), passing in the **context id**, **product id** and **service URL**. EAUserContext contains an indication of whether the customer is allowed to proceed or not; this indication is

obtained by calling Ticket.isValid(). Other methods of the ticket provide access to information about the customer and the customer's session, but the results of these methods are undefined if EAUserContext.isValid() returns false.

The getContextData() method of EAUserContext returns a hashtable with a series of name-value pairs. In this way, the application can retrieve information that is specific to it, and ignore any extra data that is not. The values are stored as Java Strings.

Note: If your game requires a database containing user information, the value for the key "UserAlias" can be retrieved from the hashtable to use as your database index. "UserAlias" is a single case and no space version of the users screen name. It is also used as the index in EA.com's database.

Java Code Sample

The following code sample, for use in browser-based games, illustrates the use of the Authorizer, AuthorizeException and Ticket classes. from the AuthClient.java file contained in the AuthClient1.5.zip.

```
// ------------------------------------------
// AuthClient.java
//
// AuthAuthIdent baseline test for
// Java client (Version 1)
//
// Author:  Steve Keller
// History:
//          08-Jun-2000 Created
//
// ------------------------------------------
// Copyright (c) 2000 EA.com
// ------------------------------------------

import java.net.*;
import java.util.*;

import com.ea.authorize.*;

public class AuthClient
{
      // ----------------------------------------
      // public constants
      public static final int SUCCESS = 0;
      public static final int EXCEPTION = -1;
      public static final int BAD_CONTEXT = -47;
      public static final int UNINITIALIZED = -255;

      // ----------------------------------------
      // instance variables
      private String results;
      private String contextID;
      private String serviceID;
      private String authSrvUrl;

      public AuthClient(String ctx, String svc, String url)
      {
            results = "";
            contextID = ctx;
            serviceID = svc;
            authSrvUrl = url;
      }

      // ------------------------------------
      // main function
      public static void main(String args[])
      {
```

```
            String ctx = "";
            String svc = "";
            String url = "";
            int val = SUCCESS;

            if (args.length < 3)
            {
                  System.out.println("Insufficient Arguments");
                  val =  UNINITIALIZED;
            }
            else
            {
                  url = args[2];
                  svc = args[1];
                  ctx = args[0];

                  AuthClient auth = new AuthClient(ctx, svc, url);
                  val = auth.doTest();

                  // show the output
                  System.out.println(auth.getResults());

            }
            // return the exit code
            System.exit(val);
      }

      // ---------------------------------
      // the test
      public int doTest()
      {
            results = "";

            int retVal = SUCCESS;

          Authorizer authorizer = new AuthorizerDefaultImpl();
          EAUserContext ctx = null;
          try
          {
            ctx = authorizer.authorize(contextID, serviceID,
authSrvUrl);
          }
          catch(AuthorizeException excp)
          {
            appendResults( "AuthorizeException while trying to
authorize context.\n" + excp.getMessage());
              retVal = EXCEPTION;
          }

            if (retVal == SUCCESS)
            {
                if (ctx.isValid())
                {
```

```
                  appendResults("  Context ID: " + ctx.getContextID() +
"\n");
                  appendResults("  User Name: " + ctx.getUserAlias() +
"\n");
                  appendResults("\n  Service ID: " + ctx.getServiceID()
+ "\n");

                  appendResults("  Data:\n");
                  Hashtable h = ctx.getContextData();
                  Enumeration enum = h.keys();

                  while (enum.hasMoreElements())
                  {
                    Object k = enum.nextElement();
                    Object v = h.get(k);
                    appendResults("    " + k.toString() + " = " +
v.toString() + "\n");
                  }
                }
                else
                {
                  appendResults("Context is invalid!\n");
                  appendResults("  Reason: " + ctx.getReasonCode());
                    retVal = BAD_CONTEXT;
                }
            }

            return retVal;
      }

      // -----------------------------------------------
      // append text to the results string
      private void appendResults(String s)
      {
            results += s;
      }

      public String getResults()
      {
            return results;
      }
}
```

C++ Implementation

To use the C++ version of the authorization client for use in browser-based games, unzip the file kcomClient1.5.zip in a directory convenient to your development environment.

This file contains the files authclient.h, authauth.lib and authauth.dll. It also contains a directory called kCOM/includes, which contains header files required by authclient.h

Include the header file authclient.h in your source. Ensure that the directory containing authclient.h and the kCOM/includes directory are visible as include directories in your project.

Link your executable against authauth.lib (or call LoadLibraryEx to explicitly load the dll). When you install your program, you must install authauth.dll and make sure your program can find it at run-time.

Clients using the Authorizer interface need to obtain three pieces of information from the execution environment: a Context id (or "**ticket"**), a **product id** and a **service URL** for the Authorization service.

The client retrieves an instance of a "ticket" by calling Authorizer.authorizeTicket(), passing in the **context id**, **product id** and **service URL**. EAUserContext contains an indication of whether the customer is allowed to proceed or not; this indication is obtained by calling Ticket.isValid(). Other methods of the "ticket" provide access to information about the customer and the customer's session, but the results of these methods are undefined if EAUserContext.isValid() returns false.

The getContextData() method of EAUserContext returns a hashtable with a series of name-value pairs. In this way, the application can retrieve information that is specific to it, and ignore any extra data that is not. The values are stored as strings.

Note: If your game requires a database containing user information, the value for the key "UserAlias" can be retrieved from the hashtable to use as your database index. "UserAlias" is a single case and no space version of the users screen name. It is also used as the index in EA.com's database.

C++ Code Sample

The following code sample, for browser-based games, is from the testAuthorization.cpp file in the kcomClient1.5.zip file.

```
// ============================================================
// authorizationTest.cpp
//
// Test code used to ensure that authorization client component
// works correctly.
//
// 3/25/00 Ed Zavada
// ============================================================


#include "authclient.h"
#include <iostream>

using namespace kcom;
using namespace ea::authorization;
using namespace std;

#ifdef K_MSVC
#pragma warning(disable:4800 4290)  // don't want MSVC's performance
warning
#endif

void TestAuthorize(char* contextid, char *serviceid, char *authSrvUrl)
throw (std::exception);

int main(int argc, char *argv[])
{
      // holders for data
      char contextid[64];
      char serviceid[64];
      char authSrvUrl[128];

      if (argc > 3)
            strcpy(authSrvUrl, argv[3]);
      else
            strcpy(authSrvUrl,
"http://gmwestdev1.ea.com:8081/authorize");

      if (argc > 2)
            strcpy(serviceid, argv[2]);
      else
            strcpy(serviceid, "GT_TEST_GAME");

      if (argc > 1)
            strcpy(contextid, argv[1]);
      else
```

```
            strcpy(contextid, "12047:42");

    try
      {
        TestAuthorize(contextid, serviceid, authSrvUrl);
    }
    catch (std::exception e) {
        cerr << "exception occurred: " << e.what();
    }

      return 0;
}

void TestAuthorize(char* contextid, char *serviceid, char *authSrvUrl)
throw (std::exception)

{
      // ------------------------------------------------------------
      //  C++ test
    cout << "Beginning Test\n\n";
    kcom::auto_interface<IAuthorizer>
auth(AuthorizationLib_GetAuthorizer());
    if (!auth.valid()) {
        cerr << "Failed to initialize AuthorizationLib!\n";
        return;
    }

    // check the server to see if the context is authorized
    kcom::Result kr;
    kcom::auto_interface<IEAUserContext> ctx( auth-
>Authorize(contextid, serviceid, authSrvUrl, kr) );

    if (kcom::ResultFailed(kr))
      {
        cerr << "Failed to Authorize EAUserContext. Result code = " <<
kr << "\n";
    }
      else if (ctx->IsValid())
      {
        cout << "user = " << ctx->GetUserAlias() << " ";
        cout << "service = " << ctx->GetServiceID() << "\n";
        kcom::auto_interface<INVPairList> seq(ctx->GetContextData());
        ea::int32 idx = seq->Reset();
        const INVPair* p;
        while ( (p = seq->Next(idx)) != 0) {
            auto_interface<const INVPair> pair(p);
            cout << pair->GetName() << " = " << pair->GetValue() <<
"\n";
        }
        const char* s = seq->GetNamedValue("name");
        if (s) {
            cout << "Looked up value for [name] is ["<<s<<"]\n";
        }
    }
```

```
      else
      {  // !ctx->IsValid()
        cerr << "Invalid User Context: " << ctx->GetReasonCode() <<
"\n";
    }
      // end C++ Test
      // -----------------------------------------------------------------


}
```

Authorization Client API's for Browser-Based Games

There are two forms of the Authorization API for browser-based games – one in Java and one in C++.

Note: The Java API is exactly the same whether an external process (such as a game client) or internal process (such as a Game Technology server) uses it.

There is no requirement that the C/C++ version of the API be provided for use by backend processes. All C/C++ clients must communicate with the Authorization servlet.

Java Authorization Client API

The Java Authorization Client API is used by Java clients to communicate with the Authorization service. The classes that form this API are all located in the Java package: **com.ea.authorize**.

Class **Authorizer**

This class is used by clients to verify a user context is valid. It has one public method:

EAUserContext **authorize** (String **contextid**, String **service_id**, String **url**)

Description:	Contact the Authorization service and present a user context id.
Inputs:	**contextid** - a Java String containing an opaque handle to this user's current game session. **service_id** – a Java String containing the EA.com identifier of the game. **url** – a Java String containing the URL of the Authorization and Authentication service.
Outputs:	EAUserContext – a class containing methods to retrieve the contents of the user context (see EAUserContext below).
Exceptions:	AuthorizeException – an exception class thrown if the service cannot be contacted for some reason. See AuthorizeException below.

Class **EAUserContext**

This class provides access to the contents of the user context retrieved by a call to Authorizer.authorize (). Access is through the following public methods:

boolean **isValid**()

Description:	Checks if user context is valid. Note that the return values of all other accessor methods, except getReasonCode(), are invalid if this function returns **false**.
Inputs:	None.
Outputs:	**true** or **false**. **true** indicates the contents of the context are valid.
Exceptions:	None.

String **getReason**Code()

Description:	Retrieves a text description of the reason the user context is invalid.
Inputs:	None.
Outputs:	A String containing the reason code. This String is empty if the EAUserContext is valid.
Exceptions:	None.

String **getUserAlias**()

Description:	Retrieves the user's screen name. It is the Formatted User Alias, with capitalization and spacing specified by the user.
Inputs:	None.
Outputs:	String containing the screen name.
Exceptions:	None.

String **getServiceID**()

Description:	Retrieves the service id that corresponds to this context.
Inputs:	None.
Outputs:	**String** containing the service id.
Exceptions:	None.

Hashtable **getContextData**()

Description:	Retrieves any application-specific information from the context. This information is in the form of name-value pairs as Strings. **Note:** If an unformatted version of the user's screen name is needed for a database index, obtain this information from the hashtable using the key "UserAlias."
Inputs:	None.
Outputs:	A **Hashtable** containing the name-value pairs.
Exceptions:	None.

Class **AuthorizeException**

This class encapsulates an exception thrown by the Authorizer.authorizet() method.

AuthorizeException (String **message**)

Description:	Constructor for the AuthorizeException class.
Inputs:	**message** – a string containing a description of the reason for the exception.
Outputs:	None.
Exceptions:	None.

String **getMessage**()

Description:	Retrieves the message description for this exception.
Inputs:	None.
Outputs:	A Java String containing the value of the message.
Exceptions:	None.

C/C++ Authorization Client API

The C/C++ client API is used by ActiveX or Netscape plug-in games, and is implemented as a kCOM shared component that can be called from both C++ and straight C.

Interface **IAuthorizer**

This is the interface used by clients to verify a user context is valid. It has one public method:

IEAUserContext* **Authorize** (const char *in**Contextid**,
const char * in**ServiceId**,
const char *in**Url,**
kcom::Result& out**Result**) throw()

Description:	Contacts the Authorization service and presents a context id.
Inputs:	**inContextid** - a pointer to a string containing an opaque handle to this user's current game session. **inServiceId** – a pointer to a string containing the EA.com identifier of the game. **inUrl** – a pointer to a string containing the URL of the Authorization and Authentication service. **outResult** – a reference to a variable of type kcom::Result which will receive the result code.
Outputs:	**IEAUserContext*** – a pointer to the interface containing methods to retrieve the contents of the user context (see **IEAUserContext** below). This interface pointer remains valid until the user calls Release() for it. We recommend that you assign it to a kcom::auto_interface<IEAUserContext> so Release() will be called for you automatically. **outResult&** – a reference to a kcom::Result code, which can be converted to an exception by calling kcom::ThrowIfFailed(), or evaluated by calling kcom::ResultSucceeded().
Exceptions:	None.

Interface **IEAUserContext**

This interface provides access to the contents of the user context retrieved by a call to IAuthorizer::Authorize (). Access is through the following public methods:

bool8 **IsValid**() const throw()

Description:	Find out if a context is valid. Note that the return values of all other accessor methods, except GetReasonCode(), are invalid if this function returns false.
Inputs:	None.
Outputs:	**true** or **false**. **true** indicates the contents of the context are valid.
Exceptions:	None.

const char* **GetReasonCode**() const throw()

Description:	Retrieve a text description of the reason the user context is invalid.
Inputs:	None.
Outputs:	A string containing the reason code. This will be an empty string (“”) if the user context is valid. This pointer is only valid until Release() is called on the EAUserContext object, so you must copy it to local storage if you want to keep it available. EA.com recommends constructing a std::string with it as follows: std::string myReason(context->GetReasonCode());
Exceptions:	None.

const char* **GetUserAlias**() const throw()

Description:	Retrieve the user's screen name
Inputs:	None.
Outputs:	A string containing the user's screen name. This pointer is only valid until Release() is called on the EAUserContext object, so you must copy it to local storage if you want to keep it available. Clients using the Authorizer interface need to obtain three pieces of information from the execution environment: a Context id (or "**ticket"**), a **product id** and a **service URL** for the Authorization service. In Java applets, these are typically passed in via the **<PARAM>** HTML tag inside the **<APPLET>** tag. The client retrieves an instance of a "ticket" by calling Authorizer.authorizeTicket(), passing in the **context id**, **product id** and **service URL**. EAUserContext contains an indication of whether the customer is allowed to proceed or not; this indication is obtained by calling Ticket.isValid(). Other methods of the ticket provide access to information about the customer and the customer's session, but the results of these methods are undefined if EAUserContext.isValid() returns false. The getContextData() method of EAUserContext returns a hashtable with a series of name-value pairs. In this way, the application can retrieve information that is specific to it, and ignore any extra data that is not. The values are stored as Java Strings. **Note:** If your game requires a database containing user information, the value for the key "UserAlias" can be retrieved from the hashtable to use as your database index. "UserAlias" is a version of the users screen name that is single case with no space between words. It is also used as the index in EA.com's database. We recommend constructing a std::string with it as follows: std::string aliasString(context->GetUserAlias());
Exceptions:	None.

const char* **GetServiceID**() const throw()

Description:	Retrieve the product id which corresponds to this context
Inputs:	None.
Outputs:	A string containing the service id. This pointer is only valid until Release() is called on the EAUserContext object, so you must copy it to local storage if you want to keep it available. We recommend constructing a std::string with it as follows: std::string aliasString(contextt->GetProductID());
Exceptions:	None.

kcom::ISequence<const INVPair>* **GetServiceID**()const throw()

Description:	Retrieve any application-specific information from the context. This information is in the form of a sequence of read only name-value pairs, accessable through the INVPair interface. **Note:** If your game requires a database containing user information, the value for the key "UserAlias" can be retrieved from the hashtable to use as your database index. "UserAlias" is a single case and no space version of the users screen name. It is also used as the index in EA.com's database.
Inputs:	None.
Outputs:	The name-value pairs as entries in a kcom::ISequence of INVPair objects.
Exceptions:	None.

Interface **INVPair**

This interface encapsulates the concept of a name-value pair of strings. It has the following public methods:

void **SetName**(const char ***name**) throw()

Description:	Sets the name part of the name-value pair.
Inputs:	**name** – the name of this name-value pair. Might be truncated if the string exceeds the internal limit.
Outputs:	None.
Exceptions:	None.

void **SetValue**(const char ***value**) throw()

Description:	Sets the value part of this name-value pair
Inputs:	**value** – the value of this name-value pair. Might be truncated if the string exceeds the internal limit.
Outputs:	None.
Exceptions:	None.

const char* **GetName**() throw()

Description:	Retrieve the name of this name-value pair.
Inputs:	None.
Outputs:	A pointer to a string containing the name. This should be copied into local storage if the application wants to keep this data for longer than the lifetime of the NVPair object, until Release() is called for that object.
Exceptions:	None.

const char * **GetValue**() throw()

Description:	Retrieve the value part of this name-value pair
Inputs:	None.
Outputs:	A pointer to a string containing the value. This should be copied into local storage if the application wants to keep this data for longer than the lifetime of the NVPair object, until Release() is called for that object.
Exceptions:	None.

Auth/Auth/Ident Standalone Games Client Usage Instructions

This section provides information on integrating the EA.com Games Technology Authorization service into standalone game client programs, such as online games or client matchmaking components. These components provide access to the Authorization service, while hiding the complexity of making a connection to the service

The delivery package consists of a single zip file, *AuthAuthIdent_v2.zip*. Unzip this file in a convenient temporary directory.

C++ In-Game Client Implementation

After unzipping the delivery zip file, you will have a file called *AuthAuthIdent_V2_GameClient.zip*. Unzip this file in a directory convenient to your game development environment.

This *AuthAuthIdent_V2_GameClient.zip* contains the header file, *gt_auth_v2.h*, the dll *GTAuthV2.dll*, the export library *GTAuthV2.lib* and a directory called kcom/includes. In this directory are some additional header files required by *gt_auth_v2.h*.

It also contains a directory called *test*, which contains an example file testauth.cpp. Use the code in this example as a guide in developing your game code.

To use this client in your code, make sure that your project can find the header files and the library. The resulting executable must be able to find the dll.

Add following three entries to your registry under:
HKEY_LOCAL_MACHINE\SOFTWARE\EACom\AuthAuth.
The keys and the current values are:
AuthLoginBaseService=AuthLogin (REG_SZ)
AuthLoginServer= ea4.str.ea.com (REG_SZ)
SSLRetry=1 (REG_DWORD)

The value of AuthLoginBaseService is the servlet name that the client will be calling. This value probably will not change, however, AuthLoginServer will certainly from time to time. SSLRetry is for use with SSL servers that have an invalid or outdated certificate. Keep this at '1' for now; this will allow the client to retry a connection in the above cases. When your game code goes into production, SSLRetry should be set to '0.'

Note: When you install your game, make sure that the previous registry entries are created on the user's machine. By the time your game ships, we will have a stable server name to put into the AuthLoginServer entry.

Game Server Library

The delivery zip file also contains a tar file, *AuthV2.tar.gz*. Extract this file to a convenient directory on your Solaris machine. It contains a header file *AuthAuthServer.h*, a shared library *libAuthAuthServer.so* and an example test file *testAuth.cpp*.

To use the library, make sure to update the LD_LIBRARY_PATH so that the library can be found. Include AuthAuthServer.h in your code and link against libAuthAuthServer.so.

You also must create and set the environment variable AUTHSERVER. The contents of this environment variable should be the name of the EA.com authorization server. Currently it is http://ea4.str.ea.com/Authorize. The path will probably change from time to time. The base server name will generally match the name in the AuthLoginServer registry entry. For example, if you are using csh or tcsh, set the environment variable as follows: setenv AUTHSERVER http://ea4.str.ea.com/Authorize.

The environment variable AUTHENSERVER must also be set. The content of this environment variable should be the name of the EA.com authentication server. Currently it is http://ea2.str.ea.com/Authenticate - this will probably change during different test phases and again this will generally match the server name of the registry entry for the C++ In-Game client. . For example if you are using csh or tcsh, , set the environment variable as follows: setenv AUTHENSERVER http://ea4.str.ea.com/Authorize.

After compiling the testAuth.cpp, the execution file has two parameters. The first one is the contextID and the second is the Game Tech Product ID. The examples of this should look like testAuth 12047:42 GT_TEST_GAME.

Embedded Game Client APIs for Standalone Games

This section presents the C++ and C embedded client API's for Auth/Auth/Ident for standalone games.

Overview

Auth/Auth/Ident for standalone games for in-game authentication consists of three parts:

- Game Client Library (described in this section)
- Game Server Library (refer to Auth/Auth/Ident Game Server Library API)
- EA.com Auth/Auth/Ident Service

The basic sequence of events for using this service is as follows:

1. Game contacts the Auth/Auth/Ident service for a "ticket"
2. Game presents the ticket to the Game Server
3. Game Server presents the ticket to the Auth/Auth/Ident service
4. Auth/Auth/Ident service returns user information to the game server

At this point, the game server has a fully logged in user.

The client library encrypts the data sent to the Auth/Auth/Ident service, so user login names and passwords remain secure. The connection between the game server and EA.com is a secure channel, so sensitive user information will never be visible on the public internet.

Use of kCOM

kCOM is a lightweight component technology invented by Kesmai Corporation. The Auth/Auth/Ident V2 client libraries use this technology to eliminate problems with recompiling and redistributing game code if changes or updates are made. All access to the library's services are through kCOM interfaces, which are available for both C and C++ clients.

C++ API

The C++ API is a little different than using a C++ class. You must first obtain a pointer to an IAuthorize interface, then use that interface to access the service. You cannot directly instantiate an IAuthorize object, nor can you delete it. The API provides a factory method for creating an instance of the interface, and the interface itself has a Release method which is used instead of deleting the object.

Factory Method

```
IAuthorize * GTAuthV2Lib_GetAuthorize()
```

Description:	Retrieves an object that implements the IAuthorize interface
Inputs:	None
Outputs:	Pointer to an object that implements the IAuthorize interface. If this object cannot be instantiated, NULL is returned.
Exceptions:	None.

Synchronous Ticket Retrieval

```
int32 GetTicketSync(const char* inUsername, const char *inPassword,
const char *inGTProductID, char *outContextID);
```

Description:	Synchronous call to retrieve a ticket
Inputs:	inUsername – the user's login name inPassword – the user's password inGTProductID – a string containing the Game Tech Product ID for the game outContextID – a pointer to a buffer to which the ticket data will be copied. The data will be an ASCII null-terminated string. This buffer must be at least 129 bytes in length (128 + 1 for null termination)
Outputs:	The result of the call will be one of the following values: IAUTHORIZE_SUCCESS IAUTHORIZE_INVALID IAUTHORIZE_BAD_CONNECTION
Exceptions:	None.

Asynchronous Ticket Retrieval

```
int32 GetTicketAsync(const char* inUsername, const char *inPassword,
const char *inGTProductID, IAUTH_FN fn)
```

Description:	Asynchronous call to retrieve a ticket.
Inputs:	inUsername – the user's login name. inPassword – the user's password. inGTProductID – a string containing the Game Tech Product ID for the game. fn – pointer to a function to receive the ticket data. See definition of the IAUTH_FN callback function below.
Outputs:	The result of the call will be one of the following values: IAUTHORIZE_SUCCESS IAUTHORIZE_INVALID IAUTHORIZE_BAD_CONNECTION
Exceptions:	None.

Callback Function

The fourth parameter to GetTicketEx is a pointer to a function that receives the ticket data. This function is defined as follows:

```
typedef void (*IAUTH_FN)(const char *);
```

Note: This function must be declared static if it is a member of a class.

C++ Code Example

Following is an example of using the C++ interface for Embedded Standalone games. Note especially the use of the factory method and the call to **`IAuthorize->Release()`**.

```
////////////////////////////////////
// testauth.cpp
//
// Test the GTAuthV2 in-game client using C++
#include <gt_auth_v2.h>

void authCallback(const char *ctx)
{
        char myCtx[129];
        strcpy(myCtx, ctx);
}

int main()
{
        char context[128];

        ea::authauth::IAuthorize *iauth = ea::authauth::GTAuthV2Lib_GetAuthorize();

        if (iauth == NULL)
                return -1;

        ea::int32 r;

        r = iauth->GetTicketSync("user", "pwd", "TEST_GAME", context);
        r = iauth->GetTicketAsync("user", "pwd", "TEST_GAME", authCallback);

        iauth->Release();

        return 0;

}
```

C API

Following is the C API for Embedded Standalone games. The C API is a very similar to the C++ API. The primary difference is that instead of using the pointer to the interface to call functions, the interface pointer is passed into the function as the first parameter.

Factory Method

This is identical to the C++ API.

Synchronous Ticket Retrieval

```
int32 IAuthorize_GetTicketSync(IAuthorize *ia, const char* inUsername,
const char *inPassword, const char *inGTProductID, char *outContextID);
```

Description:	Synchronous call to retrieve a ticket
Inputs:	ia – pointer to the IAuthorize interface. inUsername – the user's login name. inPassword – the user's password. inGTProductID – a string containing the Game Tech Product ID for the game. outContextID – a pointer to a buffer to which the ticket data will be copied. The data will be an ASCII null-terminated string. This buffer must be at least 129 bytes in length (128 + 1 for null termination).
Outputs:	The result of the call will be one of the following values: IAUTHORIZE_SUCCESS IAUTHORIZE_INVALID IAUTHORIZE_BAD_CONNECTION
Exceptions:	None.

Asynchronous Ticket Retrieval

```
int32 IAuthorize_GetTicketAsync(IAuthorize *ia, const char* inUsername,
const char *inPassword, const char *inGTProductID, IAUTH_FN fn)
```

Description:	Asynchronous call to retrieve a ticket
Inputs:	ia – pointer to the IAuthorize interface. inUsername – the user's login name. inPassword – the user's password. inGTProductID – a string containing the Game Tech Product ID for the game. fn – pointer to a function to receive the ticket data. See definition of the IAUTH_FN callback function below.
Outputs:	The result of the call will be one of the following values: IAUTHORIZE_SUCCESS IAUTHORIZE_INVALID IAUTHORIZE_BAD_CONNECTION
Exceptions:	None.

Callback Function

This is identical to the C++ API.

C Code Example

Following is an example of using the C interface for embedded standalone games. Note the call to **`IAuthorize_Release()`** at the end of this example.

```
////////////////////////////////////
// testauth.c
//
// Test the GTAuthV2 in-game client  using C

#include <stdlib.h>
#include <string.h>
#include <gt_auth_v2.h>

void authCallback(const char *ctx)
{
        char myCtx[129];
        strcpy(myCtx, ctx);
}

int main()
{
        char context[128];
        int32 r;
        IAuthorize *iauth = NULL;

        iauth = GTAuthV2Lib_GetAuthorize();

        if (iauth == NULL)
                return -1;

        r = IAuthorize_GetTicketSync(iauth, "user", "pwd", "TEST_GAME", context);
        r = IAuthorize_GetTicketAsync(iauth, "user", "pwd", "TEST_GAME", authCallback);

        IAuthorize_Release(iauth);

        return 0;
}
```

gt_auth_v2.h

The following is a complete copy of the header file you will need to use the standalone client.

```
////////////////////////////////////////
// gt_auth_v2.h
//
// Distributed header file for the
// EA.com Games Tech auth/auth/ident V2
// embedded game client library
//
// ------------------------------------
// Author:     Steve Keller
// History:
//             21-May-2000  Created
// ------------------------------------
// Copyright (c) 2000 EA.com
// ------------------------------------

#ifndef GT_AUTH_V2_H_
#define GT_AUTH_V2_H_

#include <kcom.h>

#ifdef __cplusplus

namespace ea
{
namespace authauth
{


#endif  // __cplusplus

// ---------------------------------
// declare interface id's
// ---------------------------------
enum
{
        IAuthorize_IID = 101
};

// ---------------------------------
// types used by this interface
// ---------------------------------

typedef void (*IAUTH_FN)(const char *);

// ---------------------------------
// Possible return values for functions
// in this interface
// ---------------------------------
enum
{
        IAUTHORIZE_SUCCESS = 0,
        IAUTHORIZE_INVALID = -1,
        IAUTHORIZE_BAD_CONNECTION = -2,
};
// ---------------------------------
// ---------------------------------
//             C++ interfaces
// ---------------------------------
// ---------------------------------

#ifdef __cplusplus
```

```
interface IAuthorize : public kcom::ICommonInterface
{
	enum
	{
		InterfaceID = IAuthorize_IID
	};

	method ea::int32 GetTicketSync(const char* inUsername, const char *inPassword,
const char *inGTProductID, char *outContextID) throw() = 0;
	method ea::int32 GetTicketAsync(const char* inUsername, const char *inPassword,
const char *inGTProductID, IAUTH_FN fn) throw() = 0;
};

#endif // __cplusplus

// ---------------------------------
// ---------------------------------
//             C interfaces
// ---------------------------------
// ---------------------------------

#ifndef __cplusplus

// IAuthorize methods

#define IAUTHORIZE_METHODS(_iface)    \
	KCOM_METHOD4(_iface, GetTicketSync, int32, const char* inUsername, \
const char *inPassword, const char *inGTProductID, \
char *outContextID); \
KCOM_METHOD4(_iface, GetTicketAsync, int32, const char* inUsername, \
const char *inPassword, const char *inGTProductID, IAUTH_FN fn);

// declare interface
DECLARE_KCOM_INTERFACE(IAuthorize)
	IAUTHORIZE_METHODS(IAuthorize)
END_KCOM_INTERFACE

// define interface
#define IAuthorize_GetInterface          ICommonInterface_GetInterface
#define IAuthorize_GetConstInterface     ICommonInterface_GetConstInterface
#define IAuthorize_HasInterface          ICommonInterface_HasInterface
#define IAuthorize_AddRef                ICommonInterface_AddRef
#define IAuthorize_Release               ICommonInterface_Release

#define IAuthorize_GetTicketSync(this, inUsername, inPassword, \
inGTProductID, outContextID) \
		KCOM_CALL4(this, GetTicketSync, inUsername, \
inPassword, inGTProductID, outContextID)

#define IAuthorize_GetTicketAsync(this, inUsername, inPassword, inGTProductID, fn) \
		KCOM_CALL4(this, GetTicketAsync, inUsername, inPassword, \
inGTProductID, fn)

#endif // !__cplusplus
```

```
// ------------------------------------------------------
// ------------------------------------------------------
//             Primary call to get an Authorize object.  Use instead
//             of new IAuthorize()...
// ------------------------------------------------------
// ------------------------------------------------------
KCOM_EXPORT(IAuthorize *) GTAuthV2Lib_GetAuthorize();

#ifdef __cplusplus  // close out namespaces

}       // namespace authauth
}       // namespace ea

#endif  // __cplusplus

#endif  // GT_AUTH_V2_H_
```

Linking Instructions

In addition to the header file, you need the files GTAuthV2.lib and GTAuthV2.dll. These files are the import definitions and the implementation of the library.
Include the header file at the appropriate points in your source, link against GTAuthV2.lib and deliver GTAuthV2.dll with your game. You will have to make sure that this dll is installed where your game code can find it.

Auth/Auth/Ident Game Server Library API

The AuthAuthIdent Game Server Library is used in conjunction with the standalone game client component to perform authorization of a particular user. This system ensures that a user's subscription status is current, while freeing developers from managing user accounts. You can still keep game-specific user information in your own database, but EA.com handles account management and billing

The following library has two function calls, authorize() and authenticate().

Int authorize(char *contextid, char *gtProductId, EAUserInfo *userInfo)

Description:	Performs an authorization check on a user. This function will only succeed on the first call for a particular context id.
Inputs:	**contextid** – a char * containing the contextid sent to the game server from the game client. **gtProductId** – a char * containing the Game Tech Product ID for this game **userInfo** = a pointer to a EAUserInfo structure
Outputs:	An integer describing the outcome of the call. Returned data is located in the input parameter userInfo. Valid return values defined in the file authauthserver.h: AUTH_AUTH_SUCCESS AUTH_AUTH_INVALID_TICKET AUTH_AUTH_INVALID_SERVICE AUTH_AUTH_INVALID_INPUT AUTH_AUTH_UNABLE_TO_CONNECT AUTH_AUTH_TICKET_ALREADY_USED AUTH_AUTH_MALFORMED_XML_TICKET AUTH_AUTH_EXCEPTION_CAUGHT AUTH_AUTH_FAILED
Exceptions:	None

int authenticate(char *contextid, EAUserInfo *userInfo)

Description:	Performs an authentication check on a user. This function will always succeed if the ticket part of the context id is valid, regardless of whether it has been accessed previously.
Inputs:	**contextid** – a char * containing the contextid sent to the game server from the game client. **UserInfo** – a pointer to a EAUserInfo structure.
Outputs:	An integer describing the outcome of the call. Returned data is located in the input parameter userInfo. Valid return values defined in the file authauthserver.h: AUTH_AUTH_SUCCESS AUTH_AUTH_INVALID_TICKET AUTH_AUTH_INVALID_SERVICE AUTH_AUTH_INVALID_INPUT AUTH_AUTH_UNABLE_TO_CONNECT AUTH_AUTH_TICKET_ALREADY_USED AUTH_AUTH_MALFORMED_XML_TICKET AUTH_AUTH_EXCEPTION_CAUGHT AUTH_AUTH_FAILED
Exceptions:	None

Struct EAUserInfo

```
typedef struct tag_EAUserInfo
{
	char UserAlias[32];
	char RegistrationSource[32];
	char GameTechProductID[32];
	time_t TicketCreationTime;
} EAUserInfo;
```