Principles of Applications

In the work of applications, there are the following main steps:

- Loading MT5APIManager.dll using the CMTManagerAPIFactory::Initialize method of he Manager API factory.
- Creating the manager interface using the CMTManagerAPIFactory::CreateManager method.
- Verifying that the versions of the main header file MT5APIManager.h (the version parameter of one of the interface creation methods) and of the loaded DLL (passed by CMTManagerAPIFactory::Version) match.
- Connecting an application to a server with the Connect method, using the created manager interface, as well as details of the manager account created on the server.
- After the work is completed, the application is disconnected from the server using the Disconnect method.
- Next the created interface is deleted using the Release method.
- The last stage is unloading the DLL of the manager API from the memory using the CMTManagerAPIFactory::Shutdown method.

Example of using the .NET wrapper of Manager API

```
namespace SomeClientNamespace
  {
   using MetaQuotes.MT5CommonAPI;
   using MetaQuotes.MT5ManagerAPI;
   using System;
...

   class CSomeClientClass
     {
      //--- Manager API
      public CIMTManagerAPI m_manager=null;
...
      public MTRetCode Initialize()
        {
         MTRetCode res=MTRetCode.MT_RET_ERROR;
         //--- Initialize the factory

if((res=SMTManagerAPIFactory.Initialize(@"..\..\..\API\"))!=MTRetCode.MT_RET_OK)
           {
            LogOutFormat("SMTManagerAPIFactory.Initialize failed - {0}",res);
            return(res);
           }
         //--- Receive the API version
         uint version=0;
         if((res=SMTManagerAPIFactory.GetVersion(out version))!=MTRetCode.MT_RET_OK)
           {
            LogOut("SMTManagerAPIFactory.GetVersion failed - {0}",res);
            return(res);
           }
         //--- Compare the obtained version with the library one
         if(version!=SMTManagerAPIFactory.ManagerAPIVersion)
```

```
           {
            LogOutFormat("Manager API version mismatch -
{0}!={1}",version,SMTManagerAPIFactory.ManagerAPIVersion);
            return(MTRetCode.MT_RET_ERROR);
           }
         //--- Create an instance

m_manager=SMTManagerAPIFactory.CreateManager(SMTManagerAPIFactory.ManagerAPIVersion,ou
t res);
         if(res!=MTRetCode.MT_RET_OK)
           {
            LogOut("SMTManagerAPIFactory.CreateManager failed - {0}",res);
            return(res);
           }
         //--- For some reasons, the creation method returned OK and the null pointer
         if(m_manager==null)
           {
            LogOut("SMTManagerAPIFactory.CreateManager was ok, but ManagerAPI is
null");
            return(MTRetCode.MT_RET_ERR_MEM);
           }
         //--- All is well
         LogOutFormat("Using ManagerAPI v. {0}", version);
         return (res);
        }
...
      public MTRetCode Connect(string server,UInt64 login,string password,uint
timeout)
        {
         MTRetCode res=MTRetCode.MT_RET_ERROR;
         if(m_manager==null)
           {
            LogOut(EnMTLogCode.MTLogErr,server,"Connection to {0} failed: .NET Manager
API is NULL",server);
            return(res);
           }
         //---

res=m_manager.Connect(server,login,password,null,CIMTManagerAPI.EnPumpModes.PUMP_MODE_
FULL,timeout);
         if(res!=MTRetCode.MT_RET_OK)
           {
            LogOut(EnMTLogCode.MTLogErr,server,"Connection by Managed API to {0}
failed: {1}",server,res);
            return (res);
           }
         //---
         LogOut("Connected");
         return(res);
        }
...
      //+------------------------------------------------------------------+
      //| API call example                                                 |
      //+------------------------------------------------------------------+
      private void OnRequestServerLogs(EnMTLogRequestMode requestMode,
                                       EnMTLogType        logType,
                                       Int64              from,
                                       Int64              to,
                                       string             filter=null)
        {
         if(m_manager==null)
```

```
            {
             LogOut("ERROR: Manager was not created");
             return;
            }
          //---
          LogOut(EnMTLogCode.MTLogAtt,"LogTests",CAPITester.LOGSEPARATOR);
          try
            {
             MTRetCode result=MTRetCode.MT_RET_ERROR;
             //---
             MTLogRecord[]
records=m_manager.LoggerServerRequest(requestMode,logType,from,to,filter,out result);
             //---
             LogOutFormat("LoggerServerRequest {0} ==> [{1}] return {2} record(s)",
                          (result==MTRetCode.MT_RET_OK ? "ok" : "failed"),
                          result,(records!=null ? records.Length : 0));
             //---
             if((result==MTRetCode.MT_RET_OK) && (records!=null))
               {
                foreach(MTLogRecord rec in records)
                   LogOut(rec);
               }
            }
          catch(Exception ex)
            {
             ...
            }
        }
...
     }
  }
```

Manager API Factory functions

CMTManagerAPIFactory::Initialize

```
MTRetCode  SMTManagerAPIFactory.Initialize(
   string  dll_path=NULL  // Path to DLL library of the API
   )
```

Parameters

dll_path

[in] Full path to DLL library of the Manager API.

Returned value

An indication of a successful performance is the MT_RET_OK response code. Otherwise, an error code will be returned.

Note

The method loads the DLL of the Manager API and gets addresses of exported functions. Depending on the application architecture, the 32-bit or 64-bit version of DLL is loaded.

CMTManagerAPIFactory::Version

```
MTRetCode  SMTManagerAPIFactory.GetVersion(
   out uint version    // Version
   )
```

Parameters

&version

[out] The version of the Manager API library.

Returned value

An indication of a successful performance is the MT_RET_OK response code. Otherwise, an error code will be returned.

CMTManagerAPIFactory::CreateManager

```
CIMTManagerAPI  SMTManagerAPIFactory.CreateManager(
   uint            version,       // Version
   out MTRetCode   res            // Response code
   )
```

Parameters

version

[in] Version of the MT5APIManager.h header file

manager

[out] A pointer to the manager interface IMTManagerAPI.

Returned value

An indication of a successful performance is the MT_RET_OK response code. Otherwise, an error code will be returned.

CMTManagerAPIFactory::Shutdown

```
MTRetCode  SMTManagerAPIFactory.Shutdown()
```

Returned value

An indication of a successful performance is the MT_RET_OK response code. Otherwise, an error code will be returned.

Manager API functions

Connecting to the Server

The application is connected to the platform in one of the pumping mode, which indicates what information the application will continuously receive from the server.

Pumping Mode

Enabled pumping mode means that the application will synchronize the appropriate server database with a locally created database. Thus, to obtain information the application can access a local database using the Get and Next methods (for example, IMTManagerAPI::SymbolGet, IMTManagerAPI::GroupNext, etc.). If there is no need to synchronize databases, which can be quite large, you may leave the pumping mode disabled. In this case required information can be requested directly from the server using Request methods (for example, IMTManagerAPI::SymbolRequest, IMTManagerAPI::GroupRequest).

Possible pumping modes are enumerated in IMTManagerAPI::EnPumpModes. The pumping mode in which we want to connect is specified in the pump_mode parameter of the IMTManagerAPI::Connect method.

ID	Value	Description
PUMP_MODE_USERS	0x00000001	Pumping of users.
PUMP_MODE_ACTIVITY	0x00000002	Pumping of the online activity of users.
PUMP_MODE_MAIL	0x00000004	Mail pumping.
PUMP_MODE_ORDERS	0x00000008	Pumping of orders.
PUMP_MODE_NEWS	0x00000020	News pumping.
PUMP_MODE_POSITIONS	0x00000080	Pumping of positions.
PUMP_MODE_GROUPS	0x00000100	Pumping of configurations of user groups.
PUMP_MODE_SYMBOLS	0x00000200	Pumping of symbol configurations.
PUMP_MODE_HOLIDAYS	0x00000800	Pumping of holiday configurations.
PUMP_MODE_TIME	0x00001000	Pumping of time configurations.
PUMP_MODE_GATEWAYS	0x00002000	Pumping of gateway configurations. It is used for displaying available gateways in the box of accounts in external systems on the "Account" tab of a client in the administrator or manager terminal.
PUMP_MODE_REQUESTS	0x00004000	Pumping of the request queue and results of processing requests by other dealers. It is required for connecting a manager in the "Supervisor" mode.
PUMP_MODE_PLUGINS	0x00008000	Pumping of the plugin configurations on the server. It is required for the possibility of setting up plugins through the Manager API application. Additionally the plugin must have the IMTConPlugin::PLUGIN_FLAG_MAN_CONFIG flag enabled, and the manager account must have the IMTConManager::RIGHT_CFG_PLUGINS permission granted.
PUMP_MODE_FULL	0xffffffff	Pumping of all the information listed above.

IMTManagerAPI::Connect

```
MTRetCode  CIMTManagerAPI.Connect(
   string                        server,           // Server address
   ulong                         login,            // Login
   string                        password,         // Password
   string                        password_cert,    // Certificate password
   CIMTManagerAPI.EnPumpModes    pump_mode,        // Pumping mode
   uint                          timeout           // Timeout
   )
```

Parameters

server

[in] Colon separated address and port of the trading server to which you want to connect. The address and port of the access server are specified here.

login

[in] The login of the manager for connection.

password

[in] The password of the manager for connection.

password_cert

[in] Certificate password. This parameter is used only for the advanced authorization mode.

pump_mode

[in] The pumping mode, in which connection will be established. To pass the pumping mode, the IMTManagerAPI::EnPumpModes enumeration is used.

timeout=INFINITE

[in] Server connection timeout in milliseconds. If the application fails to connect to a server within this time, attempts will be terminated. By default there is no time limit.

Returned value

An indication of a successful performance is the MT_RET_OK response code. Otherwise, an error code will be returned.

Note

After establishing a connection to the server, Manager API will automatically keep it up. In case of connection loss, Manager API will automatically reconnect to the server.

IMTManagerAPI::Subscribe

Subscribe to common events of the IMTManagerAPI interface.

```
MTRetCode  CIMTManagerAPI.Subscribe(
   CIMTManagerSink  sink      // CIMTManagerSink object
   )
```

Parameters

sink

[in] A pointer to the object that implements the IMTManagerSink interface.

Returned value

An indication of a successful performance is the MT_RET_OK response code. Otherwise, an error has occurred that corresponds to the response code.

Note

Subscribing to events is thread safe. One and the same interface IMTManagerSink cannot subscribe to an event twice - in this case the response code MT_RET_ERR_DUPLICATE is returned.

The object at which sink points, must remain in the memory (must not be removed) until the call of Unsubscribe or until the administrator interface is deleted using the IMTManagerAPI::Release method.

IMTManagerAPI::Unsubscribe

Unsubscribe from common events of the IMTManagerAPI interface.

```
MTRetCode  CIMTManagerAPI.Unsubscribe(
   CIMTManagerSink  sink      // CIMTManagerSink object
   )
```

Parameters

sink

[in] A pointer to the object that implements the IMTManagerSink interface.

Returned value

An indication of a successful performance is the MT_RET_OK response code. Otherwise, an error code will be returned.

Note

This method is pared to IMTManagerAPI::Subscribe. If an attempt is made to unsubscribe from the interface to which it has not subscribed, MT_RET_ERR_NOTFOUND error is returned.

IMTManagerAPI::Disconnect

```
void  CIMTManagerAPI.Disconnect()
```

Sink object interfaces

IMTManagerSink

IMTManagerSink::OnConnect

A handler of the event of an application's connection to the server.

```
virtual void  CIMTManagerSink.OnConnect()
```

Note

The method is called when the connection of the manager or administrator interface with the server has been established/restored.

IMTManagerSink::OnDisconnect

A handler of the event of an application's disconnection from the server.

```
virtual void  CIMTManagerSink.OnDisconnect()
```

Note

The method is called when the connection of the manager or administrator interface with the server is lost.

IMTDealSink

IMTDealSink::OnDealAdd

A handler of the event of adding a deal.

```
virtual void  CIMTDealSink.OnDealAdd(
   CIMTDeal        deal      // Deal object
   )
```

Parameters

deal

[in] A pointer to the object of the added deal.

Note

This method is called by the API to notify that a new deal has been added.

Data structures

IMTUser

IMTUser::Login

Get the login of a user.

```
ulong  CIMTUser.Login()
```

Returned value

The login of a user.

IMTUser::Group

Get the group to which the user is included.

```
string  CIMTUser.Group()
```

Returned value

If successful, it returns a pointer to a string with the group of a user. Otherwise, it returns NULL.

Note

The pointer to the resulting string is valid for the lifetime of the IMTUser object.
To use the string after the object removal, a copy of it should be created.

IMTUser::Balance

Get the current balance of a client.

```
double  CIMTUser.Balance()
```

Returned value

The current balance of a client.

IMTDeal

IMTDeal::Deal

Get the ticket of a deal.

```
ulong  CIMTDeal.Deal()
```

Returned value

Deal ticket.

IMTDeal::Action

Get the type of action performed with a deal.

```
uint  CIMTDeal.Action()
```

Returned value

A value of the IMTDeal::EnDealAction enumeration.

IMTDeal::EnDealAction

Types of actions performed by a deal are enumerated in IMTDeal::EnDealAction.

ID	Value	Description
DEAL_BUY	0	A Buy deal.
DEAL_SELL	1	A Sell deal.

IMTDeal::Symbol

Get the symbol, for which a deal was executed.

```
string  CIMTDeal.Symbol()
```

Returned value

If successful, it returns a pointer to a string with the symbol name and path to it in accordance with the hierarchy of symbols in the platform. Otherwise, it returns NULL.

Note

The pointer to the resulting string is valid for the lifetime of the IMTDeal object.
In order to use the string after the object removal, a copy of it should be created.

IMTDeal::Volume

Get the volume of a deal.

```
ulong  CIMTDeal.Volume()
```

Returned value

The deal volume in the UINT64 format (one unit corresponds to 1/10000 lot).

IMTDeal::Profit

Get the value of the profit from the deal execution.

```
double  CIMTDeal.Profit()
```

Returned value

Profit from a deal.

IMTDeal::TimeMsc

Gets the time of a deal execution in milliseconds.

```
long  CIMTDeal.TimeMsc()
```

Returned value

The time of a deal execution in milliseconds that have elapsed since January 1, 1970.

IMTDeal::PositionID

Gets the position ID (ticket) specified in the deal.

```
ulong  CIMTDeal.PositionID()
```

Returned value

The position ID set in the deal.

Note

The position ID is used for analyzing the history of working with the position. A unique identifier PositionID is assigned to all orders and deals that open, modify and close this position, as well as to the position itself. This identifier corresponds to the ticket of the order, the execution of which resulted in position opening.
The identifier is not changed when a position is reversed in the netting mode.

Application Requirements

When developing applications, it is necessary to meet the following requirements:

- An application should be as efficient in memory usage as possible.
- An application should fragment memory as little as possible.
- An application should not cause memory leaks.
- An application must quickly return control from event handlers.

Implementation of MetaTrader 5 Manager API in .NET

The MetaTrader 5 Manager API includes a ready implementation in .NET. The implementation is available in the form of ready to use DLLs. To be able to use their functions, you need to add them to a project.

- MetaQuotes.MT5CommonAPI64.dll — library of common API interfaces (configuration bases and databases).
- MetaQuotes.MT5ManagerAPI64.dll — Manager API library.

No Exceptions

Managed wrappers do not generate exceptions. If any library calls generate exceptions, the wrapped code will not let them out.

Register Event Handler Classes, Do Not Allow Exceptions

For sink classes, you must call the RegisterSink method in the child class constructor and check the return value. This is connected with the possibility to process errors binding a native sink interface to its managed wrapper.
It is important to not let exceptions out of the sink classes handler methods. Otherwise, logging exceptions will be impossible. It is especially important not to let the exceptions out of the hook methods, which may cause unpredictable behavior of the trading platform.

Always Use Factory

All managed wrappers of API interfaces should only be created using factories, special methods (for example, UserCreate, UserCreateAccount, etc). This will prevent from passing incorrect pointers to the native code and from using unsafe client code. A separate factory is implemented for each API library.

Use the "using" Operator

All API managed wrappers inherit the IDisposable interface, which provides a mechanism for releasing unmanaged resources. Use the "using" operator for easy and correct use of IDispose objects.

```
void UserFunction()
  {
...
   using(CIMTUser user=m_manager.UserCreate())
     {
      user.Name("<UserName>");
...
      user.Leverage(leverage);
...

m_manager.UserAdd(user,master_password,investor_password);
     }
     //--- Will surely destroy the user object and its native
resources
  }
```

In the above example, the 'user' object and its native resources will be released once code execution exits the 'using' block. Without using the "using" operator, it is unknown when and under what circumstances the object will be deleted.
To explicitly release resources, you can also use the 'Release' and 'Dispose' methods of managed wrappers. These methods are similar.

ToArray Methods in Managed Wrappers for Working with Arrays

ToArray methods are implemented in all managed wrappers for working with array objects, such as CIMTUserArray, CIMTOrderArray etc. These methods copy the elements of the source array to a new array of the same type.

- ToArray creates copies of elements of the source array, so the array returned by ToArray can be used after the source array is released.
- If the source array does not have elements, ToArray returns an empty array, not nullptr.