SimbaProvider SDK 4.6

Developer's Guide for C++

October 6, 2016

Simba Technologies Inc.

Simba Technologies Inc.

938 West 8th Avenue
Vancouver, BC Canada
V5Z 1E5

Tel. +1 (604) 633-0008
Fax. +1 (604) 633-0004

www.simba.com

Printed in Canada

Table of Contents

Table of Figures

Introduction

Welcome to the SimbaProvider for OLAP SDK™ 4.6 Developers Guide. This guide describes the basic development tasks necessary to take the sample OLAP provider in the SDK and use it as a starting point in building your own provider.

Note: This guide assumes that you already have the SimbaProvider for OLAP SDK™ installed and configured for development as described in the SimbaProvider for OLAP SDK™ 4.6 Quick Start Guide. It also assumes that you have a good understanding of OLAP principles and terminology.

Development of a new provider is usually done by taking the sample provider included in the SDK and modifying it to access your own data source. Version 4.6 of the SimbaProvider for OLAP SDK™ includes a sample data source implemented using .csv (text) files. This allows you to get up and running quickly with the sample since no additional drivers or software (e.g. servers) are necessary. Note that prior to SimbaProvider for OLAP SDK™ 4.5, the sample data required SQL Server, Microsoft Analysis Services, and other components to be installed. Existing users who have followed these steps in the past will therefore need to review the new sample code described in this document to understand any required changes.

Developing Your Provider in C++

Copying and Modifying the ODBO Sample Provider as a Starting Point

This section describes how to take the sample ODBO provider in the base kit, copy it, and modify it so that it can be used as a starting point in creating your own ODBO provider.

1. In Visual Studio, open the *<Install Path>/SimbaProvider for OLAP SDK 4.6/SampleProvider/Win32/Provider/Main/SimbaProvider VS2015.sln* solution and use the *Build | Clean Solution* menu to remove excess files.

   Note: In the evaluation version of the kit, the solution file is named: *SimbaProvider_Eval VS2015.sln*.
2. Close the solutions.
3. Create a new directory on your hard drive. This directory will be the root of your provider source code tree (referred to as <*YourSourceRootFolder*>).
4. Copy and paste the *SampleProvider* and *Simba* folders to this new directory.
5. Update the **SIMBAHOME** environment variable to point to <*YourSourceRootFolder*> (the root of your provider source code tree). This environment variable specifies the directory that contains the Simba folder.

6. Choose an appropriate name for your new provider and rename the *SampleProvider* folder copied in the previous step. From now on, we will refer to this new folder as *<YourProviderFolder>*.
7. (Optional) Rename *SimbaProvider VS2015.vcxproj* located in*<YourProviderFolder>/Win32/Provider/Main/* to match the name chosen in Step 6 (e.g. TestProvider.vcxproj). You will also need to perform the following additional steps:

    - Open the *.vcxproj* file in a text editor and replace all instances of *SimbaProvider* with the base portion of the new .vcxproj file name (e.g. TestProvider).
    - Open the *SimbaProvider VS2015.sln* file in a text editor and replace all instances of SimbaProvider base portion of the new *.vcxproj* file name (e.g. TestProvider).
    - Open *SimbaProvider.def* and replace the instance of SimbaProvider after the *LIBRARY* keyword with the new name chosen.
    - (Optional) Open *SimbaProvider.rc* and replace the hardcoded string resource values as you see fit (for example, CompanyName).
    - Rename *SimbaProvider.rc* and *SimbaProvider.def* to match the new name (e.g. TestProvider.rc and TestProvider.def).

8. (Optional) Rename SimbaProvider VS2015.sln also located in *<YourProviderFolder >/Win32/Provider/Main/* to match the name chosen in Step 6 (e.g. TestProvider.vcxproj).

    We will refer to your renamed *SimbaProvider VS2015.sln* file as *<YourProviderSolutionFile>* and the project your renamed *SimbaProvider VS2015.vcxproj* file refers to as *<YourProviderProject>* from now on.
9. (Optional) Rename *<YourProviderFolder >/Common/SampleProviderVersion.h* based on the name chosen in Step 6. The contents of this file can be modified at any time to adjust the version numbers for your provider. If you choose to rename this file, you will also have to update the #includes in the following files to refer to the new filename:

    - *<YourProviderFolder>/Win32/Provider/Main/Version.h*
    - *<YourProviderFolder>/Win32/Provider/SimErrorLookup/Version.h*

    Note the following:

    - *SampleProviderVersion.h* is also used in the XMLA provider solution that will require similar updates to accommodate a new file name.
    - The XMLA ProviderVersion property is populated from the *SAMPLEPROVIDER_FRIENDLYVERSION* value. Some applications such as Tableau or Reporting Services will read this value when connecting to XMLA using their Analysis Services connector. To properly support these applications, the major number of the *SAMPLEPROVIDER_FRIENDLYVERSION* value must be greater than or equal to the Analysis Services version supported by the application. In practice,

this means that the major number must be 9 or higher. However, this may change as applications and implementations migrate to newer Analysis Services editions.

The list below shows the Analysis Services edition and corresponding version number.

- 2005 – 9
- 2008 – 10
- 2008R2 – 10.5
- 2012 – 11
- 2014 – 12
- 2016 - 13

10. Open <YourProviderSolutionFile> in Visual Studio.
11. Open *Version.h* from *<YourProviderProject>* and modify the values for the VER_FILENAME_STR and G_szProviderDllName #defines to reference the new name of your .dll. Optionally modify the other #define values as you see fit. The string values defined here appear in the data source information property set.
12. (Optional) Rename and modify the SimErrorLookup output DLL filename using the new name chosen in Step 6:
    a. Close the solution.
    b. Open Version.h in <YourProviderFolder>\Win32\Provider\SimErrorLookup in a text editor and replace all instances of SimbaProviderEr with the new name of your DLL (e.g. TestProviderEr).
    c. Open SimErrorLookup VS2015.vcxproj and replace all instances of SimbaProviderEr with the new name (e.g. TestProviderEr).
    d. Open ScErrLkp.def and replace the instance the SimbaProviderEr on the first line with your new DLL base name (TestProviderEr).
    e. (Optional) Open ScErrLkp.rc and modify the hardcoded string resources as you see fit (for example, CompanyName).
13. Create new GUIDs for both the provider and error lookup projects:
    a. Navigate to *Tools* | *Create GUID* in Visual Studio to open the GUID generation tool.
    b. Select "2. DEFINE_GUID(...)".in the GUID Format group box.
    c. Click *New GUID*.
    d. Click "Copy" to copy the newly generated GUID to the clipboard.
    e. Open *Guid.h* in *<YourProviderProject>*.
    f. Paste the GUID generated in Step C in place of the *Provider GUID* code-block.
    g. Rename *CLSID_SimbaProvider* based on the new name for your provider (e.g. CLSID_TestProvider).
    h. Repeat the previous steps to generate new GUIDs in place of the *ProviderErrorInfo GUID* code block and for the *Property Page GUIDs*. (The property page CLSID does not have to be renamed.)
    i. Rename *CLSID_SimbaProviderErrorInfo* based on the new name for your provider (e.g. CLSID_TestProviderErrorInfo).

  j. Replace all instances of *CLSID_SimbaProvider* and *CLSID_SimbaProviderErrorInfo* in the solution with the new names created in Steps g and i.

14. Rebuild the solution.

You now have a working ODBO provider that provides access to the underlying OLAP data stored in .csv files. The following sections will describe the implementation of key elements of the sample provider, and provide suggestions on how to modify them for your own data source.

Manually Registering your ODBO Provider

Since an ODBO provider is a COM-based technology, it must be registered in the Windows registry on each system for which it is to be used. In the previous section, the sample ODBO provider automatically registered the provider on your local development machine in the *Post-Build Event* defined in the configuration of the *SimbaProvider* project.

If you plan to deploy your provider on another machine, it will need to be registered on that machine either manually or through an installer that you provide to your customers.

The registration process will vary slightly depending on whether the provider and the target platform that it is intended for is 32-bit or 64-bit. The registry in 64-bit versions of Windows contains two registries— the main 64-bit registry and a 32-bit registry hive that runs as a windows-on-windows (*WOW*) process (refer to MSDN for more details). Correspondingly, you need to use a different *regsvr32.exe* program to register your 32-bit and 64-bit provider. The following cases show how to register your provider based on its bitness and that of your OS:

1. **32-bit provider on a 32-bit OS or 64-bit provider on a 64-bit OS**: run regsvr32.exe.
2. **32-bit provider on a 64-bit OS**: run regsvr32.exe found in <Windows directory>\SysWOW64.

Note: Windows requires that *regsvr32.exe* be run as administrator.

Copying and Modifying the .NET XMLA Sample Provider as a Starting Point

This section describes how take the sample .NET XMLA provider in the base kit, copy it, and modify it so that it can be used as a starting point in creating in your own XMLA provider. The .NET XMLA provider generates a web service for use in IIS. This section assumes that you are familiar with setting up a web service in IIS and therefore does not contain the specific steps necessary.

1. In Visual Studio, open the *<Install Path>/SimbaProvider for OLAP SDK 4.6/Sample Provider/dotNET/XmlaProvider VS2015.sln* solution and use the *Build | Clean Solution* menu to remove excess files.

   Note: In the evaluation version of the kit, the solution file is named: *XmlaProvider_Eval VS2015.sln*.

2. Close the solutions.
3. Create a new directory on your hard drive. This directory will be the root of your provider source code tree.
4. Update the **SIMBAHOME** environment variable to point to the root of your provider source code tree. This environment variable specifies the directory that contains the Simba folder.
5. Copy and paste the *SampleProvider* and *Simba* folders to this new directory.
6. Choose an appropriate name for your new provider and rename the *SampleProvider* folder copied in the previous step. From now on, we will refer to this new folder as *<YourProviderFolder>*.
7. (Optional) Rename the solution and project files:
    a. Rename *XmlaProvider VS201x.sln* (or *XmlaProvider_Eval VS201X.sln*) based on the new name chosen in Step 5 (e.g. *TestProvider.VS2015.sln*)
    b. Rename Customer.ManagedIso VS201x.vcproj in *<YourProviderFolder>\SampleProvider\dotNET\ISO.NET\*. It is recommended that the *.ManagedIso* portion remain to help identify this project (e.g. *TestProvider.ManagedIso.vcproj*).
    c. Open the renamed *Customer.ManagedIso.vcproj* file in a text editor and replace all instance of *Customer.ManagedIso* with the base portion of the project file name (e.g. *TestProvider.ManagedIso*).
    d. Open *<YourProviderFolder>\SampleProvider\dotNET\XMLA VS201x\ CustomerXmlaWebService.csproj* in a text editor and replace all occurrences of *Customer.ManagedIso* with the base portion of the project file name (e.g. TestProvider.ManagedIso).
    e. Change all occurrences of *CustomerXmlaWebService* based on the new name (e.g. TestProviderXmlaWebService.
    f. Rename CustomerXmlaWebService.csproj with the base portion of the project file name (e.g. *TestProviderXmlaWebService.csproj*).
    g. Open the solution file located in *<YourProviderFolder>\SampleProvider\dotNET/* using a text editor and replace all occurrences of *Customer.ManagedISO* based on the new name (e.g. TestProvider.ManagedIso). Also, replace all occurrences of *CustomerXmlaWebService* in that file with the base portion of the project file name (e.g. *TestProviderXmlaWebService*).
    h. Open the solution in Visual Studio.
    i. Open *<YourProviderFolder>\dotNET\XMLA VS201x\Properties\AssemblyInfo.cs* and replace all occurrences of *CustomerXmlaWebService* based on the new name chosen (e.g. *TestProviderXmlaWebService*).
    j. (Optional) Change the name of the *XmlaWebService.asmx* file within your renamed *CustomerXmlaWebService* project. This file is the endpoint in the URL that consumers will navigate to when connecting to your provider. Note that this step is not necessary and you can change the file name later at any point.

   k. (Optional) Rename *<YourProviderFolder>\Common\SampleProviderVersion.h* and edit the version information in that file. Update all occurrences in the solution where this file is #included with the new filename.
   l. Update the company information in *AssemblyInfo.cpp* from the renamed *Customer.ManagedIso* project as well as the information in *AssemblyInfo.cs* from the renamed *CustomerXmlaWebService project.*

   Note: The DLL created by your renamed *Customer.ManagedIso* project will not have a version tab when you view the DLL properties in Windows Explorer. This is because the Visual C++ compiler does not generate a Win32 VERSIONINFO resource from the assembly manifest. If you want a version tab to appear when you view the DLL properties in Windows Explorer, you will need to create a resource file containing the VERSIONINFO resource and add it to your renamed *Customer.ManagedIso* project. You will need to update the resource file manually to keep version numbers in sync.

8. (Optional if not done when modifying ODBO) Rename *<YourProviderFolder >/Common/SampleProviderVersion.h* based on the name chosen in Step 6. The contents of this file can be modified at any time to adjust the version numbers for your provider. If you choose to rename this file, you will also need to update the #includes in the following files to refer to the new filename:

   - *<YourProviderFolder>/Win32/Provider/Main/Version.h*
   - *<YourProviderFolder>/Win32/Provider/SimErrorLookup/Version.h*

   Note: SampleProviderVersion.h is also used in the XMLA provider solution, which will require similar updates to accommodate a new file name.

9. (Optional if not done when modifying ODBO) Open *Version.h* from *<YourProviderProject>*. Optionally, modify the other #define values as you see fit. The string values defined here appear in the data source information property set. Open *SampleProvider.h* and update the version numbers as appropriate.

10. Open *DataSources.xml* within the renamed *CustomerXmlaWebService* project and update the *<URL>* field to point to the renamed .asmx file, ensuring that the alias for the virtual directory that you plan to use is correct (e.g. *http://localhost/TestXmlaWebServiceEval/XmlaWebService.asmx*).

11. (Optional) Update the values in *DataSources.xml* for the *<DataSourceName>*, *<DataSourceDescription>*, and *<ProviderName>* fields as required.

12. Build the solution.

13. Create a web service in IIS and set its physical path to that of the newly generated text provider web service: *<YourProviderFolder>\dotNET\XMLA VS201x*.

The provider can now be used as a web service to provide access to the underlying OLAP data stored in .csv files. The following sections will describe the implementation of key elements of the provider, and provide suggestions on how to modify them for your own data source.

Basic Provider Implementation Tasks

The following subsections provide enough information to take the sample provider and modify it to perform the following basic tasks:

- Return the supported capabilities of your provider
- Provide the ability to connect to your data source
- Set the current catalog
- Reset the data source to its initial state
- Expose the supported syntax of your provider
- Clean up the data source
- Handle and communicate errors to the user

This set of tasks is considered the minimal set of functionality required for a new provider. Once you have completed these tasks, you can move onto the next major section for more advanced tasks.

Introduction to the ClassFactory and Data source classes

The entry point into the provider is the *ClassFactory* which contains methods for returning information about the provider (i.e. its capabilities) and for creating a data source. The data source works with an underlying data store and provides the calling framework with methods to access data.

The sample provider's *Datasource* class uses a text file data store called *TextOlap* that stores data in a set of .csv files. The sample *Datasource* class therefore contains the minimal information necessary to provide access to this text-based data store. The following list describes each of the members of the *Datasource* class:

- **m_pConnectionInfo**: caches connection related information that is passed in by the calling framework when it attempts to connect to the data source.
- **m_isConnected**: stores the state of the connection to the underlying data store. This will be set to true when a successful connection is made. Since the sample data store consists of text files, the connection will always be successful.
- **m_refCount**: used for managing references to the data store.
- **m_currentCatalog**: stores the name of the current catalog.
- **m_catalogLock**: used for synchronizing access to *m_currentCatalog* from different threads.
- **m_pServerDefaultCatalog**: stores the name of the data store's "default" catalog.

The following sections describe the various elements of the sample provider which must be implemented when creating your own provider.

Returning Provider Information

One of the first elements to implement in the provider is the method which returns information about the provider and its capabilities to the calling framework, and ultimately to the consumer application (e.g. Excel). Since you are writing your own provider, you will want to modify the information returned to encompass the details specific to your provider.

The code to perform this exists in the ISO implementation's *ClassFactory* class which contains a method called *GetProviderInfo()* that returns a *CSOProviderInfo* object containing information about the provider (for example, the provider name, version and capabilities).

The *ClassFactory::GetProviderInfo()* method can therefore be considered the entry point into your provider and thus one of the first methods to modify when creating your own provider so that you can return your own information. The following snippet shows the method's signature:

```
void ClassFactory::GetProviderInfo(
      Simba::ISO::CSOProviderInfo& providerInfo,
      Simba::ISO::ESOPROVIDERINFO ulProviderInfoID)
{
      .
      .
}
```

This method will be invoked multiple times by the calling framework, each time requesting specific information via the *ulProviderInfoID* parameter. The *ulProviderInfoID* parameter is an enumeration of type ESOPROVIDERINFO (defined in *ISODatabase.h* in the SimbaProvider OLAP SDK™) which specifies the information to be returned.

Some of the information requested by the calling framework will be requested regardless of the type of provider, while other information is specific to ODBO and XMLA (e.g. a GUID will only be requested in the case of an ODBO (COM-based) provider, since XMLA providers do not contain a GUID).

For simplicity the sample implementation of *ClassFactory::GetProviderInfo()* (shown in the following snippet) ignores the parameter and always returns all information (note that some information is only returned for ODBO (i.e. the code in #ifnef XMLA)). In your own provider, it would be good practice to examine the value passed in for *ulProviderInfoID* and return only the information requested, especially if your code is only being built for one type of provider (i.e. ODBO or XMLA).

```
void ClassFactory::GetProviderInfo(
      Simba::ISO::CSOProviderInfo& providerInfo,
      Simba::ISO::ESOPROVIDERINFO ulProviderInfoID)
{
      // Get the customer-specified cache size.
      unsigned long cacheSize =
      CustomerGlobalVariables::GetInstance().GetCacheSize();

      // Properties of the provider.

      #ifndef XMLA
      Simba::ISO::CSOGUID guid1( CLSID_SimbaProvider ) ;
      providerInfo.SetGuid( &guid1 ) ;
      #endif
      // This name is visible to end-users/consumers
      providerInfo.SetProviderName( G_szProviderName ) ;
      providerInfo.SetAsynchSupport( true ) ;
      providerInfo.SetProviderFriendlyName( G_szProviderFriendlyName );
      providerInfo.SetProviderDLLName( G_szProviderDLLName ) ;
      providerInfo.SetProviderVersion( G_szProviderVersion ) ;
      providerInfo.SetCacheSize( cacheSize ) ;
      // Same as MSOLAP.
      providerInfo.SetDefaultPrompt( DBPROMPT_NOPROMPT ) ;

      providerInfo.SetPropertyPageSupport( true );
      #ifndef XMLA
      Simba::ISO::CSOGUID connectionGuid(CLSID_ConnectionPropertyPage);
      Simba::ISO::CSOGUID advancedGuid(CLSID_AdvancedPropertyPage);
      providerInfo.SetPropertyPageConnectionClsid(&connectionGuid);
      providerInfo.SetPropertyPageAdvancedClsid(&advancedGuid);
      #endif

      // Properties of the errorinfo.
      #ifndef XMLA
      Simba::ISO::CSOGUID guid2( CLSID_SimbaProviderErrorInfo ) ;
      providerInfo.SetErrorGuid( &guid2 ) ;
      providerInfo.SetErrorName( G_szErrorName ) ;
      #endif

}
```

Note: Some of the information returned in *GetProviderInfo()* has been hard coded into #defines located in the ISOImpl's *Version.h* header file. This information can be modified as required by your provider.

The following table provides information about all of the possible values that can be requested via *ulProviderInfoID* and the types of providers that the value will be called for.

ESOPROVIDERINFO Values	**Description**	**Provide Type(s)**
PROVIDERINFO_GUID	The provider's GUID. Use the *SetGuid()* method to set this value.	ODBO

PROVIDERINFO_ERRORGUID	A GUID for the error lookup DLL you are writing. Use the *SetErrorGuid()* method to set this value.	ODBO
PROVIDERINFO_NAME	The name of your provider. Use the *SetProviderName()* method to set this value.	ODBO and XMLA
PROVIDERINFO_ERRORNAME	The name of your error lookup DLL. Use the *SetErrorName()* method to set this value.	ODBO
PROVIDERINFO_FRIENDLYNAME	A description of your provider. Use the *SetProviderFriendlyName()* method to set this value.	ODBO and XMLA
PROVIDERINFO_DLLNAME	The file name of your provider DLL. Use the *SetProviderDLLName()* method to set this value.	ODBO
PROVIDERINFO_VERSION	The current version of your provider. Use the *SetProviderVersion()* method to set this value.	ODBO and XMLA
PROVIDERINFO_SUPPORTSASYNCH	A Boolean value (true, false) that indicates whether your provider supports asynchronous processing. Use the *SetAsynchSupport()* method to set this value.	ODBO and XMLA
PROVIDERINFO_CACHESIZE	An integer value that indicates whether the ODBO layer will perform caching of cells. A value of 0 turns caching off, otherwise it represents the number of cells that the ODBO layer will cache. Use the *SetCacheSize()* method to set this value. Note: Please set this to 0 if the MDX engine is running on the same machine as the ODBO layer (which is usually the case)	ODBO
PROVIDERINFO_DEFAULTPROMPT	An integer value that can be used to set the ODBO layer's default value for the DBPROP_INIT_PROMPT property. This is useful for avoiding problems when consumer applications do not set this property. Use the *SetDefaultPrompt()* method to set this value.	ODBO

In the sample implementation, the *GetProviderInfo()* method also enables "property pages" for ODBO using the following code:

```
providerInfo.SetPropertyPageSupport( true );
#ifndef XMLA
Simba::ISO::CSOGUID connectionGuid(CLSID_ConnectionPropertyPage);
Simba::ISO::CSOGUID advancedGuid(CLSID_AdvancedPropertyPage);
```

```
providerInfo.SetPropertyPageConnectionClsid(&connectionGuid);
providerInfo.SetPropertyPageAdvancedClsid(&advancedGuid);
#endif
```

Property pages allow the end user to perform configuration of the provider at runtime such as specifying a data source. This is useful for cases where the consumer application does not already have the necessary information or credentials to connect to a data source.

In the example above, two property pages are enabled when the code is built as part of an ODBO provider; one page specifying a connection and another "advanced" page that contains some sample controls for demonstration purposes:

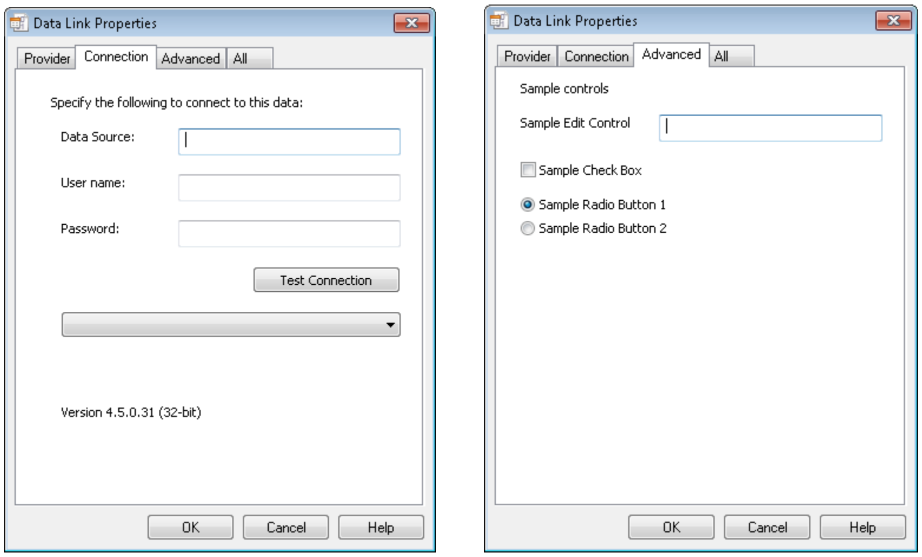


Figure 1 - Property Pages Enabled by the Sample Provider

After the user enters this information, the calling framework then tries to retrieve the specified data source from your provider. The provider's data source is therefore the next element of the provider to address and is described in detail in the next section.

Connecting to the Data Source

The previous section described how the *ClassFactory::GetProviderInfo()* method is the entry point into your provider and is used to provide information to the calling framework about the provider. In this section, we will look at how a connection to a data source is established.

After invoking the *ClassFactory::GetProviderInfo()* method, the calling framework will then request access your provider's data source by calling the *ClassFactory::CreateDataSource()* method and passing in information via the *pConnectionInfo* parameter. The following snippet shows the method's signature:

```
Simba::ISO::ISODatasource * ClassFactory::CreateDatasource(
      Simba::ISO::CSOConnectionInfo * pConnectionInfo) const
{
      .
```

```
        .
}
```

The *CreateDataSource()* method is responsible for creating and connecting to a *ISODatasource* object using the information provided, and then returning a reference to that object which the calling framework can then use in subsequent API calls to access your data. An *ISODatasource* implementation in turn, provides all of the code necessary for the calling framework to access your OLAP data store. Therefore addressing *ClassFactory::CreateDatasource()* is the second step necessary for implementing your own provider.

ISODatasource is an interface defined by the SimbaProvider for OLAP SDK. The sample provider's implementation of this interface is a class called *Datasource* (see *Datasource.cpp* and *Datasource.h)* located in the ISO implementation project.

The following snippet shows the sample provider's *CreateDataSource()* method which constructs a new instance of *Datasource*, tells it to connect to the underlying data store using the information provided by the calling framework, and then returns a reference to the data source:

```
Simba::ISO::ISODatasource * ClassFactory::CreateDatasource(
      Simba::ISO::CSOConnectionInfo * pConnectionInfo) const
{
      // Nothing to protect -- no locking required.

      CLASS_ENTER_EX_W(L"ClassFactory", L"CreateDatasource", NULL) ;

      Simba::Utils::AutoReleasePtr<Datasource> pDatasource ( new
            Datasource()) ;

      pDatasource->Connect( *pConnectionInfo ) ;

      CLASS_EXIT_EX_W( L"ClassFactory", L"CreateDatasource", NULL) ;
      return pDatasource.Detach() ;
}
```

To simplify the implementation of a data source, the sample provider's *Datasource* class uses a helper class provided in the SimbaProvider for OLAP SDK called the *DataStoreAPI*. This API exists in the *DataStoreAPI* singleton class (see *DataStoreAPI.h*) and defines a simplified set of methods for connecting and disconnecting to a data store and for performing common OLAP operations such as retrieving a list of catalogs or retrieving cell data, for example. The *DataStoreAPI* class in turn delegates all of these calls out to a *DataStoreAPIImpl* class that interacts directly with the data. The sample provider's *TextOlap* project contains an implementation called *DataStoreText* that works with text (.csv) files.

Note: The *DataStoreAPI* is an internal API that Simba uses to connect to any test data store. It is commonly used by evaluation users to implement a data store that can be used by their data source. Release users and advanced users typically use the SDK's full set of APIs.

The sample provider's *Datasource::Connect()* method uses the *DataStoreAPI* singleton to connect to the data store and authenticate the user as follows:

```
void Datasource::Connect(
        Simba::ISO::CSOConnectionInfo & connectionInfo   // in, out
)
{
        .
        .
        .
        DSStatus dsStatus =

        Simba::DataStoreAPI::DataStoreAPI::GetInstance().ConnectToServer(
                connectionInfo.GetDataSourceName(),
                connectionInfo.GetProtocolSequenceStr()
        ) ;

        if (dsStatus != S_OK )
        {
                CustomerError err( dsStatus ) ;
                throw Simba::ISO::CSOCommonException(
                        err.ErrorMessageUnicode() ) ;
        }
        else
        {
                m_isConnected = true ;
        }

        dsStatus =

        Simba::DataStoreAPI::DataStoreAPI::GetInstance().DataStoreAuthent
                icateUser(
                        const_cast<wchar_t *>( connectionInfo.GetUserID() ),
                        const_cast<wchar_t *>( connectionInfo.GetPassword() )
                ) ;

        if (dsStatus == DB_SEC_E_AUTH_FAILED)
                throw Simba::ISO::CSOAuthFailedException() ;
        else if (dsStatus != S_OK)
        {
                CustomerError err( dsStatus ) ;
                throw Simba::ISO::CSOCommonException(
                        err.ErrorMessageUnicode() ) ;
        }

        .
        .
        .
}
```

When developing your own provider, you will need to create your own *DataStoreAPImpl* implementation. How the data source is connected depends on the requirements of specific data sources. Information passed through the *CSOConnectionInfo* object (in *DataSource::Connect()*) is used to connect to the data source.

The sample's *DataStoreText* implementation of *ConnectToServer()* and *DataStoreAuthenticateUser()* methods simply return a status of *S_OK* since there is no connection or authentication involved with text files:

```
DSStatus DataStoreText::ConnectToServer(
     const wchar_t * machineName,
     const wchar_t * protocolString)
{
     // This function attempts to establish a connection to the Test
     // Server on the given host machine.
     return S_OK ;
}

DSStatus DataStoreText::DataStoreAuthenticateUser(
     /*[in]*/   const DSString  userName,
     /*[in]*/   const DSString  password)
{
     return S_OK;
}
```

Note: On Windows, *S_OK* and other return values are defined in *WinError.h*, which is part of the Windows development SDK. On Linux these have been defined in *Platform.h*, located in *<YourProviderFolder>\Simba\Linux\Utils*.

Prompting for Connection Information

When developing an ODBO provider, there may be some cases when the calling framework may not have collected enough information from the user before calling your *Datasource::Connect()* method. In this situation, there will not be enough information to establish a connection successfully. This can occur for example, when the user enters a wrong password.

Therefore you should always call *CSOConnectionInfo::GetPrompt()* on the *connectionInfo* parameter passed into *Connect()* to determine if the user should be prompted for more information.

The *GetPrompt()* method will return one of the following values:

- **DBPROMPT_PROMPT**: always prompt the user for initialization information.
- **DBPROMPT_COMPLETE**: prompt the user only if more information is needed.
- **DBPROMPT_COMPLETEREQUIRED**: prompt the user only if more information is needed. Do not allow the user to enter optional information.
- **DBPROMPT_NOPROMPT**: do not prompt the user. In this case, the *connectionInfo* parameter contains all of the information necessary to connect, so an attempt should be made to connect using this information. If this connection fails then an exception should be thrown.

Note: The sample provider does not illustrate this check for prompting. It always forces no prompt by calling *providerInfo.SetDefaultPrompt(DBPROMPT_NOPROMPT)*.

Note: These values have been defined in *<YourProviderFolder>\Simba\Common\Provider\ISO\ISODatabase.h* but originate from the Microsoft OLE DB SDK - see http://msdn.microsoft.com/en-us/library/windows/desktop/ms714342%28v=vs.85%29.aspx for more information about these values.

Note that prompting does not apply when building an XMLA provider since it runs as a web service on a server, as opposed to an ODBO provider which runs as a component on the user's local machine. If you're building an provider for XMLA, then your *Datasource::Connect()* method should never call *GetPrompt()*. Instead it should call the various methods of the *connectionInfo* parameter to get the supplied information, and then determine if there is information missing. If information is missing, then your implementation of *Datasource::Connect()* must throw an exception, thereby forcing the user to enter the information on the property pages.

If you are developing an ODBO provider and *GetPrompt()* indicates that prompting for more information is necessary, then you must provide a dialog box which is constructed and populated with values stored in the *connectionInfo* parameter using the following methods of *CSOConnectionInfo*:

- **GetInstance()**: returns the instance of the provider DLL.
- **GetWindow()**: returns the handle of the window to be used to prompt for additional information.
- **GetDatasourceName()**: returns the name of the data source entered by the user, or NULL if nothing was entered.
- **GetUserID()**: returns the user ID entered by the user, or NULL if nothing was entered.
- **GetPassword()**: returns the password of the data source entered by the user, or NULL if nothing was entered.
- **GetInitialCatalogName()**: returns the catalog name selected by the user, or NULL if nothing was selected.
- **GetProvStr()**: returns provider-specific connection information.

Any information the user enters in the dialog should override the information stored in the *connectionInfo* parameter. Use the equivalent 'Set' methods of *CSOConnectionInfo* to replace any information modified by the user in your dialog box prompt.

Returning the Data Source's Capabilities

One of the first few calls made to your provider by the calling framework is to request information about your provider's capabilities. Therefore, the next step in developing your own provider is to tell the calling framework about its capabilities.

Some examples of capabilities that can be specified include the number of active sessions, a flag indicating if the data source is read only, and an indication if flattening is supported. Specifying capabilities is necessary because consumer applications like Excel may change their behavior based on the capabilities of your provider (e.g. send different types of queries).

This information is made available to the calling framework through your provider's *Datasource::GetInfo()* method, which takes in a bit mask of *ESOINFO* enumerations specifying the capabilities that information is being requested for. A reference to a *CSOInfo* object containing the information is then returned. In the sample provider's *Datasource::GetInfo()* implementation shown below (shortened for illustration purposes), the method checks the *ulInfoID* parameter, creates a *CSOInfo* object, and stores the appropriate information in the newly created object:

```
void Datasource::GetInfo(
      Simba::ISO::CSOInfo& info,
      Simba::ISO::ESOINFO ulInfoID) const
{

      if (ulInfoID & Simba::ISO::INFO_MDPROPERTYSET)
      {
            Simba::ISO::ISO_MDPROPERTYSET    ISOMDPropSet ;

            ISOMDPropSet.bDataSourceReadOnly = false ;
            ISOMDPropSet.bMdxQueryByProperty = false ;
            ISOMDPropSet.iActiveSessions     = 0 ;
            ISOMDPropSet.iAxes               = 3 ;
            ISOMDPropSet.iMdxNamedSets       = 15;
            ISOMDPropSet.iCatalogUsage       =
                  DBPROPVAL_CU_DML_STATEMENTS ;
            ISOMDPropSet.iDsoThreadModel     = DBPROPVAL_RT_FREETHREAD ;
            ISOMDPropSet.iFlatteningSupport  = MDPROPVAL_FS_FULL_SUPPORT;
            ISOMDPropSet.iIdentifierCase     = DBPROPVAL_IC_UPPER ;
            ISOMDPropSet.iMdxAggregateCellUpdate  =
                  MDPROPVAL_AU_UNSUPPORTED ;

            other properties set here…
            .
            .
            .

            info.SetMDPropertySet( &ISOMDPropSet ) ;
      }

      if (ulInfoID & Simba::ISO::INFO_RSOPTIONSUPPORT)
      {
            Simba::ISO::ISO_RSOPTIONSUPPORT ISORSOptionSupport ;

            ISORSOptionSupport.bCatalog= true ;
            ISORSOptionSupport.bSchema = false ;

            info.SetRSOptionSupport( &ISORSOptionSupport ) ;
      }

      if (ulInfoID & Simba::ISO::INFO_RSPROPERTYSET)
      {
            Simba::ISO::ISO_RSPROPERTYSET ISORSPropSet ;

            ISORSPropSet.bCacheDeferred           = false ;
            ISORSPropSet.bCanHoldRows               = true ;
            ISORSPropSet.bCanFetchBackwards         = false ;
            ISORSPropSet.bCanScrollBackwards        = false ;
            ISORSPropSet.bDeferred                  = false ;
            ISORSPropSet.bOtherInsert               = true ;
            ISORSPropSet.bOtherUpdateDelete         = true ;
            ISORSPropSet.bQuickRestart              = false ;
            ISORSPropSet.bReturnPendingInserts      = false ;
            ISORSPropSet.bRowset                    = true ;
            ISORSPropSet.iAccessOrder               =
                  DBPROPVAL_AO_RANDOM;
            ISORSPropSet.iBookMarkInfo              = 0 ;
            ISORSPropSet.iCommandTimeOut            = 0 ;
            ISORSPropSet.iDataSourceType            = DBPROPVAL_DST_MDP ;
            ISORSPropSet.iMaxOpenRows               = 0 ;

            info.SetRSPropertySet( &ISORSPropSet ) ;
      }

      if (ulInfoID & Simba::ISO::INFO_OPTIONALINTERFACE)
      {
            Simba::ISO::ISO_OPTIONALINTERFACE ISOOptionalInterface ;

            ISOOptionalInterface.bInternal   = true ;
            ISOOptionalInterface.bSupported  = false ;

            info.SetOptionalInterface( &ISOOptionalInterface ) ;
      }

      if (ulInfoID & Simba::ISO::INFO_SESSIONPROPERTYSET)
      {
            Simba::ISO::ISO_SESSIONPROPERTYSET ISOSessionPropSet ;
            ISOSessionPropSet.iAutoCommitIsolationLevel =
                  DBPROPVAL_TI_READCOMMITTED ;
            ISOSessionPropSet.iDefaultIsolationLevel    =
                  DBPROPVAL_TI_READCOMMITTED ;
            ISOSessionPropSet.bAutoStartTransaction     = false ;
            info.SetSessionPropertySet( &ISOSessionPropSet ) ;
      }
}
```

Note: The ISO_RSOPTIONSUPPORT structure and the related code in the snippet above are discussed in further detail in "Retrieving Metadata" on page 26.

The fields in the *ISOMDPropSet*, *ISORSPropSet*, *ISOOptionalInterface*, and *ISOSessionPropSet* structures correspond to those originally defined by OLE DB, though they apply to both ODBO and XMLA. For more information see the following links:

- http://msdn.microsoft.com/en-us/library/windows/desktop/ms723066%28v=vs.85%29.aspx
- http://msdn.microsoft.com/en-us/library/ee301572.aspx#endNote105

Setting the Current Catalog

After successfully connecting to your data source, the calling framework will specify, by name, which catalog in your data store is to become the "current" one, by calling the *SetCurrentCatalog()* method of your data source. This information will be retrieved at a later point by the calling framework, via the data source's *GetCurrentCatalog()* method.

Since the sample provider's data comes from a small set of text (.csv) files, it simply caches the name passed in as shown in the following snippet:

```
void Datasource::SetCurrentCatalog( const wchar_t * pCatalogName )
{
     // This modifies the "current catalog" property for the
     // datasource. This must be thread-safe.

     Simba::Utils::AutoLock lock( &this->m_catalogLock ) ;

     this->m_currentCatalog = pCatalogName ;
}
```

For more advanced data sources that you might build your provider around, additional logic may need to be added such as to retrieve the information from the data store itself or another location.

Resetting the Datasource

In XMLA, a data source is expected to be used in a connection pool (i.e. to allow the data source to be used by different connections). To support safe reuse, the provider's *Datasource* class includes a *Reset()* method which is invoked by the calling framework to get the data source back to its initial state.

In the case of the sample provider, the only task to perform here is to ensure that the current catalog is set to that which was specified at the start of the session. In the snippet below, this is accomplished by retrieving the initial catalog name stored in the connection information (*m_pConnectionInfo*) that was cached when the connection was established, and setting it as the current catalog:

```
bool Datasource::Reset()
{
     // Resets the datasource object. For the Sample Provider, the
     // only work that needs to be done is to reset the current
     // catalog back to the initial catalog.

     this->SetCurrentCatalog( this->m_pConnectionInfo->
          GetInitialCatalogName() ) ;
     return true ;
}
```

When implementing your own provider, you will need to modify *Datasource::Reset()* to perform any additional relevant tasks for resetting your data source's state. This could include tasks like resetting the underlying data store or clearing any cached values, for example.

Returning Metadata for the Supported Syntax

An OLAP data provider must be able to return the syntax that it is capable of handling, including supported literals and keywords. This information is known as metadata and consumer applications using OLAP providers will query for this metadata by issuing the OLE DB call (through *IDBInfo::GetLiteralInfo()*) or XMLA *Discover* command and specifying the type of metadata to return.

An OLAP provider will in turn return a *schema rowset* containing the requested data. However, the construction of this rowset is done for you by the SDK when it invokes the methods described in the following two subsections.

Returning the Supported Literals

An OLAP provider must be able to return the list of string literals that it supports when a consumer application requests it. Examples of string literals include catalog names and cube names, for example. The supported literals should be returned in a rowset with the schema defined by *DISCOVER_LITERALS. The* rowset contains rows in the following columns describing information about each literal:

- **Literal ID**: a unique ID specifying the type of literal. These are defined by the SDK in *<YourProviderFolder>\Simba\Common\Provider\ISO\ISODatabase.h*. The IDs start with DBLITERAL_ (e.g. DBLITERAL_CATALOG_NAME).
- **Literal Value**: the actual value of the literal, if applicable.
- **Invalid Chararacters**: an array of characters which are not supported by the literal.
- **Invalid Starting Characters**: an array of characters which the literal cannot start with.
- **Maximum Length of Chararacters**: the maximum number of characters that the literal can be composed of.

Behind the scenes, the data source's *GetLiteralInfo()* method will be invoked when the command is issued and is therefore the method in which the data for this rowset is constructed. This method takes in a reference to a vector that can store *CSOLiteral* objects. A *CSOLiteral* object contains the fields that correspond to each of the columns described above.

The sample provider's *GetLiteralInfo()* method hard codes the literals that it supports as shown below. The first set that it prepares to return are the literals of common OLAP elements (for example, catalog or cube). Each of these literals set the invalid characters and invalid starting characters returned by the underlying MDX engine:

```
void Datasource::GetLiteralInfo(
     std::vector<Simba::ISO::CSOLiteral>   & literals // out
) const
```

```
{
     // Many of the literal types all boil down to identifiers anyway,
     // so they will all contain roughly the same information.
     // Therefore, we start with a list of all literals we plan to
     // support.
     DBLITERAL supportedIdentifierLiteralIDs[] = {
          DBLITERAL_CATALOG_NAME,
          DBLITERAL_CUBE_NAME,
          DBLITERAL_DIMENSION_NAME,
          DBLITERAL_HIERARCHY_NAME,
          DBLITERAL_LEVEL_NAME,
          DBLITERAL_MEMBER_NAME,
          DBLITERAL_PROPERTY_NAME
     } ;

     int numSupportedIdentifierLiteralIDs =
          sizeof(supportedIdentifierLiteralIDs) / sizeof(DBLITERAL);

     // For each identifier literal, create a CSOLiteral using
     // information obtained from the MDX Engine and add it to the
     // output vector.
     Simba::Utils::AutoReleasePtr<Simba::MDXEngine::IdentifierDescriptor>
          pIdentifierDescriptorAuto(
               Simba::MDXEngine::Engine::GetIdentifierDescriptor() );

     for (int i = 0 ; i < numSupportedIdentifierLiteralIDs ; i++)
     {
          TString invalidChars = pIdentifierDescriptorAuto->
               GetInvalidCharacters();
          TString invalidStartingChars = pIdentifierDescriptorAuto->
               GetInvalidStartingCharacters();

          literals.emplace_back(
               supportedIdentifierLiteralIDs[i],
               L"",
               Simba::Utils::UnicodeTools::TStringToUnicode(
          invalidChars ),
               Simba::Utils::UnicodeTools::TStringToUnicode(
          invalidStartingChars ),
               -1   // No size limit.
          ) ;
     }
     .
     .
     .
```

The sample then configures the remaining literals that it supports individually, to specify their unique configurations. Note that for the quote character prefix and suffix literals, the method fetches the character from the underlying MDX engine:

```
     .
```

```
        .
        .
        // The remaining literals must be hard-coded.
        literals.emplace_back(
            DBLITERAL_CATALOG_SEPARATOR,
            L".",
            L"",
            L"",
            0    // Length not relevant.
        ) ;

        literals.emplace_back(
            DBLITERAL_TEXT_COMMAND,
            L"",
            L"",
            L"",
            0    // Length not relevant.
        ) ;

        literals.emplace_back(
            DBLITERAL_USER_NAME,
            L"",
            L"",
            L"",
            0    // Length not relevant.
        ) ;

        TString beginQuoteChars    = pIdentifierDescriptorAuto->
            GetBeginQuoteCharacters() ;
        TString endQuoteChars = pIdentifierDescriptorAuto->
            GetEndQuoteCharacters() ;

        literals.emplace_back(
            DBLITERAL_QUOTE_PREFIX,
            Simba::Utils::UnicodeTools::TStringToUnicode(
        beginQuoteChars ),
            L"",
            L"",
            0    // Length not relevant.
        ) ;

        literals.emplace_back(
            DBLITERAL_QUOTE_SUFFIX,
            Simba::Utils::UnicodeTools::TStringToUnicode(endQuoteChars)
        ,
            L"",
            L"",
            0    // Length not relevant.
        )
    }
```

In your implementation of *Datasource::GetLiteralInfo()*, you will need to return an appropriate set of *CSOLiteral* info objects based on that literals that your provider supports.

Returning Supported Keywords

The provider must also expose all of the keywords that are recognized by the provider in MDX text commands. A consumer application will query for this information by executing the OLE DB call (*IDBInfo::GetLiteralInfo()*) or XMLA *Discover* command and specifying that the *DISCOVERY_KEYWORDS* rowset be returned.

This information is exposed through the data source's *GetKeywords()* method which takes in a reference to a vector for storing the keyword strings. Therefore the *returned* rowset consists of just one column of metadata containing keyword strings.

In the sample provider, the *GetKeywords()* method calls out to the ISM's *FunctionFactory* and *MethodFactory* to get all of their functions as well as all of the keywords supported by the MDX engine. It then adds all of the function names and keywords to the string vector passed into *GetKeywords()*:

```
void Datasource::GetKeywords(
     std::vector<std::wstring> & keywords  // out
) const
{
     // Get all customer-implemented functions and methods first.
     const Customer::ISM::FunctionFactory & functionFactory =
          Customer::ISM::FunctionFactory::GetInstance() ;

     const Customer::ISM::MethodFactory & methodFactory =
          Customer::ISM::MethodFactory::GetInstance() ;

     Customer::ISM::FunctionInfoVector mdxFunctions ;

     functionFactory.ListFunctions( L"", L"", false, false,
          mdxFunctions ) ;
     methodFactory.ListMethods( L"", L"", false, false, mdxFunctions) ;

     // Now get all keywords reported by the MDX Engine.
     std::vector<std::wstring> keywordsFromEngine ;

     Simba::MDXEngine::Engine::GetMDXKeywords( keywordsFromEngine ) ;

     // Add all keywords, functions, and methods to an STL set. By
     // adding them to a set, we automatically sort them and eliminate
     // duplicates.
     std::set<std::wstring,
          Simba::Utils::CaseInsensitiveUnicodeStringLessThan>
          keywordSet ;

     for (unsigned int i = 0 ; i < mdxFunctions.size() ; i++)
     {
          std::wstring functionNameUnicode =
               Simba::Utils::UnicodeTools::TStringToUnicode(
                    mdxFunctions[i]->GetFunctionName()
               ) ;

          keywordSet.insert( functionNameUnicode ) ;
     }

     for (unsigned int j = 0 ; j < keywordsFromEngine.size() ; j++)
     {
          keywordSet.insert( keywordsFromEngine[j] ) ;
     }

     // Everything should now be sorted and all duplicates eliminated.
     // Time to output all the keywords to the caller's vector.
     std::set<std::wstring,
          Simba::Utils::CaseInsensitiveUnicodeStringLessThan>::iterat
          or itr;

     for (itr = keywordSet.begin() ; itr != keywordSet.end() ; itr++)
     {
          keywords.push_back( *itr ) ;
     }
}
```

When implementing your provider you will need to determine all words including function and method names which are to be reserved for your provider and return them via *Datasource::GetKeywords()*. Note however, that full implementation of this method requires the ISM to be implemented first so that the supported function and method names can be returned from the ISM's *FunctionFactory* and *MethodFactory* objects. When first starting to implement your provider, use the sample implementation above, until you implement your own ISM.

Cleaning up the Data Source

The data source must implement a destructor that disconnects from the data store and performs any clean up tasks such as freeing memory.

The sample provider's *Datasource* destructor (shown below) illustrates these tasks. It starts by checking if it is in a connected state and if so, tells the data store to disconnect. It then frees up the *CSOConnectionInfo* object and as well as the string containing the name of the default catalog that it cached during startup:

```
Datasource::~Datasource()
{
      // Disconnect from the server.
      if (m_isConnected)
      {

      Simba::DataStoreAPI::DataStoreAPI::GetInstance().Disconnect() ;
      }

      if (m_pConnectionInfo != NULL)
```

```
        {
            m_pConnectionInfo->Release();
            m_pConnectionInfo = NULL;
        }

        if (m_pServerDefaultCatalog != NULL)
        {
            Simba::DataStoreAPI::DataStoreAPI::GetInstance().FreeString
                (m_pServerDefaultCatalog);
            m_pServerDefaultCatalog = NULL;
        }
    }
```

Handling Errors

Use the information provided in this section, when your provider implementation needs to notify the user about "real world" errors that occur (for example, failure to connect).

The SimbaProvider for OLAP SDK includes an exception class called *CSOException* (defined in *<YourProviderFolder>\Simba\Common\Provider\ISO\ISODatabase.h*) along with a number of derived exceptions, which must be thrown by your code when errors occur.

The use of the *CSOException* class and its derived exceptions ensures that an error is properly handled by the calling framework regardless of whether you are developing an ODBO or XMLA provider. The following points describe the underlying differences in error handling between the two, which are being handled for you by the SDK:

- **ODBO error handling**: ODBO providers must report all errors to the consumer by returning an HRESULT value that is specific to Windows. Therefore if you are developing an ODBO provider, *CSOException* will automatically map to the appropriate HRESULT once thrown to the calling framework. The mapping is done by the ODBO library.
- **XMLA error handling**: since XMLA providers exist as web services, they return results and errors to remote client consumer applications through the following XML responses:
    - **Discover/Execute errors**: if a *Discover* or *Execute* command sent to a provider causes an error, a provider must return a *SOAP fault* response containing details about the error including:

      Fault Code: set to *soap:Server* if the error occurred due to a problem on the server, or set to *soap:Client* if the Discover/Execute command sent by the consumer caused the problem.
      Fault String: a detailed description of the problem.
      Fault Actor: the URL that the consumer application used to access the provider.
      Detail: contains any application-specific error information related to the *Body* element.

      Note: No data or partial result set is returned in this case.

      The following snippet shows an example of a SOAP fault:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
     <soap:Body>
     <soap:Fault>
     <faultcode>soap:Client</faultcode>
     <faultstring>The request contains an error. ---&gt;
          Unknown set identifier: "[BadSet]"
     </faultstring
     <faultactor>http://localhost/CustomerXmla/XmlaWebServic
          e.asmx
     </faultactor>
     <detail />
     </soap:Fault>
     </soap:Body>
</soap:Envelope>
```

- **Result failures**: if the cells of a dataset or rowset returned by the provider contain an error, the error message will be embedded directly in the data result set a shown in the following snippet:

```
<Cell CellOrdinal="0">
<Value>
<Error>
<Description> An integrity constraint violation or data
     conversion error was detected while accessing a field in
     the current row.
</Description>
</Error>
</Value>
<FmtValue> Infinite recursion </FmtValue>
</Cell>
```

The usage of the *CSOException* class and those derived from it will automatically perform the necessary logic to propagate errors without having to deal with HRESULTS, SOAP faults, and/or embedding XMLA errors. It is all handled for you. This allows you to throw the same exceptions regardless of whether your provider is targeting ODBO, XMLA or both.

Note: Good programming practice dictates that exceptions, and not assertions, be used to catch errors which occur due to "real world" events such as an invalid password, or inability to connect to a data store. The use of exceptions allows a provider implementation to perform cleanup and other restorative procedures before the consumer application and end user are notified of the error. Assertions on the other hand must only be used for development purposes to ensure that code is robust.

Each of the exceptions derived from *CSOException* is intended to report a specific error (e.g. *CSOQueryException* should be thrown when an error in the command text send to your provider is detected). You may also want to derive your own exceptions for other errors that your provider needs to encounter. To so do, simply derive a class from *CSOException* and ensure it contains a constructor that takes in a *wstring* that will hold the error message to display. The following snippet shows the implementation of *CSOQueryException:*

```
class CSOUnsupportedMethodException : public CSOException
{
     // Thrown when a call to an unsupported optional method is
     // called.

public:

     CSOUnsupportedMethodException()
          : CSOException(
               ISO::ProvideMessageDescFactory().UnsupportedMethodMsg() )
     {
          ; // Do nothing.
     }

     CSOUnsupportedMethodException( const std::wstring& message )
          : CSOException( message )
     {
          ; // Do nothing.
     }

} ;
```

The *CSOException* class is only to be used within an ISO implementation. To throw an exception within an ISM implementation, the ISMException class (defined in *<YourProviderFolder >\Simba\Common\Provider\ISM\Error\ISMException.h*) or one of its derived exceptions must be thrown. This allows the MDX engine to filter errors that it may be able to handle, from those that will be sent to the ISO layer.

Retrieving Metadata

A provider must return not only its capabilities but also metadata. The following subsections describe this in more detail.

Exposing Support for Catalogs and Schemas

Once your basic provider is running, the next step is to specify whether the provider supports the *DBSCHEMA_CATALOGS* OLAP schema rowset.

The section on *Returning the Data Source's Capabilities*, introduced the *Datasource::GetInfo()* method that is used for communicating the provider's capabilities to the calling framework and it is in this method where support for these rowsets is specified.

The following snippet taken from the sample implementation of the *Datasource::GetInfo()* method specifies that the *DBSCHEMA_CATALOGS* rowset is supported by setting the *bCatalog* field to *true*:

```
void Datasource::GetInfo(
     Simba::ISO::CSOInfo& info,
     Simba::ISO::ESOINFO ulInfoID
) const
{
     .
     .
     .
     if (ulInfoID & Simba::ISO::INFO_RSOPTIONSUPPORT)
     {
          Simba::ISO::ISO_RSOPTIONSUPPORT ISORSOptionSupport ;

          ISORSOptionSupport.bCatalog= true ;
          ISORSOptionSupport.bSchema = false ;

          Info.SetRSOptionSupport( &ISORSOptionSupport ) ;
     }
}
```

Therefore, depending on the capabilities of your provider, you may need to modify the Boolean value passed into *bCatalog*.

Note: Another schema rowset called *DBSCHEMA_SCHEMATA* exists in some providers. No known ODBO providers implement DBSCHEMA_SCHEMATA, and the XMLA specification has no equivalent concept. For these reasons, we ignore DBSCHEMA_SCHEMATA in the SimbaProvider for OLAP SDK and this is why *bSchema* is set to false in the snippet above. It is only mentioned here for completeness.

Constructing the Columns for the Supported Schema Rowsets

The next step is to construct the columns for the various schema metadata rowset tables which will be later populated by the provider and returned to the calling framework when requested.

In the sample provider, the construction of column information is performed in *SchemaTableDescriptor::CreateColumnInfo()* (located in *<YourProviderFolder>\SampleProvider\Win32\Provider\ISOImpl\TableDescriptors.cpp*).

One instance of the *SchemaTableDescriptor* class will be created by the sample provider for each schema rowset, so the *CreateColumnInfo()* method checks the table GUID assigned to it and then constructs an *ISOColumnInfoVector* containing the collection of column descriptors for the table.

The sample implementation creates columns for all of the basic schema rowsets that all providers will handle, however if your custom provider needs to handle other schema rowsets, then *SchemaTableDescriptor::CreateColumnInfo()* will need to be modified.

Modifying the Session

The next step is to review and update some of the methods in the *Session* object (*<YourProviderFolder>\SampleProvider\Win32\Provider\ISOImpl\session.cpp*). The session enables a provider's data source to serve data based on the consumer's request. The session can therefore be considered the main interface between your provider and the calling framework.

In the sample provider, the *Session* object contains a number of pieces of information to maintain its state such as a session key (session identifier) and connection information, for example. A *Session* also works directly with a data store that also maintains session information so that it knows the session to which to serve data. Therefore the *Session* object is responsible for not only maintaining state information but ensuring that it coordinates session state changes with the underlying data store.

The *Session* object is also the interface where the calling framework retrieves schema rowset information from.

Maintaining Session State

To properly maintain session state, the *Session* object's constructor, destructor, and *Connect()* method need to be examined and updated.

Constructor

The *Session* object's constructor must ensure that all members are initialized correctly. One or more session objects will be created throughout the lifetime of the provider by the provider's *Datasource::CreateSession()* method. Depending on the requirements of your data source to form and maintain a connection with your data store, you may need to modify the constructor to take in and store additional variables.

The following snippet shows the implementation provided in the sample that takes in the minimal set of information (references to connection information and the data source):

```
Session::Session(
      const Simba::ISO::CSOConnectionInfo * pConnectionInfo, // in
      Datasource * pDatasource                               // in, own
): m_pSessionInfo (new Customer::ISO::CommonSessionInfo())
{
      m_pConnectionInfo     = pConnectionInfo ;
      m_SessionKey          = 0 ;
      m_pDatabase           = NULL ;
      m_pResolution         = NULL ;
      m_refCount            = 1 ;
      m_isCommandCancelled  = false;

      m_pParentDatasource   = pDatasource ;
      m_pParentDatasource->AddRef() ;

      m_sessionCatalog = m_pParentDatasource->GetCurrentCatalog() ;

      std::wstring userId = std::wstring(this->m_pConnectionInfo->
           GetUserID());
      m_pSessionInfo->SetSessionInfo(USERNAME, &userId);
}
```

Destructor

The *Session* object's destructor is responsible for deleting and cleaning up the session. The sample implementation is shown here:

```
Session::~Session()
{
     if (m_pParentDatasource != NULL)
     {
          m_pParentDatasource->Release() ;
          m_pParentDatasource = NULL ;
     }

     // This call might fail, but there's nothing that can be done
     // about it in a destructor.
     Simba::DataStoreAPI::DataStoreAPI::GetInstance().DataStoreDestroy
          Session(m_SessionKey);

     _DestroyCaches() ;
}
```

This implementation releases its pointer to the data source that was referenced in the constructor, and then tells the *Datastore API* to destroy any session state information, followed by a call to clean up cache information (this can be ignored for now, as it is an ISM detail).

Of particular interest is the call to *Simba::DataStoreAPI::DataStoreAPI::GetInstance().DataStoreDestroySession()* which must be replaced with a corresponding call into your own data store implementation to tell your data store that state information for the session is to be removed.

In the sample provider, the session key maintained by the *Session* object identifies each particular session, and is obtained from the underlying data store when a connection to that data store is established (this will be described below in the *Connect()* method). The key is used in subsequent calls to the data store so that the data store can identify the session making the calls.

Since the sample's data store consists of a collection of .csv files, there is no logic to perform so it will ignore the key and simply return *S_OK* in all cases.

Connect() Method

When a connection to a data source is requested, the Session's *Connect()* method will be invoked. It is in this method that the session tells the underlying data store to create a session as shown in the following snippet from the sample provider:

```
void Session::Connect()
{
     DSStatus dsStatus =

     Simba::DataStoreAPI::DataStoreAPI::GetInstance().DataStoreCreateS
ession(
          const_cast<wchar_t *>( m_pConnectionInfo->GetUserID() ),
            const_cast<wchar_t *>( m_pConnectionInfo->GetPassword() ),
            &m_SessionKey
        ) ;


    .
    .
    .

    }
```

In this example, the user credentials necessary for a data store connection are passed in along with a member to hold the session's identifier (session key) which will be assigned by the data store, to identify the session. In the sample provider this is always set to -1 by the text provider since separate session information does not need to be maintained for data originating in .csv files.

In your provider implementation, you will need to change this method to make the appropriate calls and pass the appropriate information to your particular data store to notify it about the new session.

Returning Schema Table Metadata and Data

When the calling framework requests schema rowsets, it invokes a number of methods in the provider to retrieve column information, rowset restrictions, and of course the rowset itself, populated with metadata.

Returning the list of Schema Rowsets Supported by the Provider

A provider must tell the calling framework about the rowsets that it supports along with the supported restrictions. By default, the sample provider already informs the calling framework that it supports the following rowsets: *DBSCHEMA_CATALOGS*, *DBSCHEMA_SCHEMATA*, *MDSCHEMA_CUBES*, *MDSCHEMA_DIMENSIONS*, *MDSCHEMA_HIERARCHIES*, *MDSCHEMA_LEVELS*, *MDSCHEMA_MEASURES*, *MDSCHEMA_PROPERTIES*, and *MDSCHEMA_MEMBERS*.

When implementing your own provider, you must inform the calling framework about any additional schema rowsets that your provider supports by modifying the *Session::GetSchemas()* method. Currently the sample provider informs the calling framework that it also supports *MDSCHEMA_FUNCTIONS*, *MDSCHEMA_SETS*, *MDSCHEMA_MEASUREGROUPS*, *MDSCHEMA_MEASUREGROUP_DIMENSIONS*, and *MDSCHEMA_KPIS* by returning a vector of rowset GUIDs as follows:

```
Simba::ISO::CSOGUIDVector * Session::GetSchemas(
      unsigned long ** ppRestrictionSupport,     // out, own
      std::vector<std::wstring>  &  schemaNames  // out
) const
{

      try
      {
          .
          .

          // Add the names.
          schemaNames.push_back( L"MDSCHEMA_FUNCTIONS" ) ;
          schemaNames.push_back( L"MDSCHEMA_SETS" ) ;
          .

      }

      if (ppRestrictionSupport == NULL)
          throw Simba::ISO::CSOInvalidArgException() ;

      // We need to output the GUIDs of all extra rowsets (only
      // FUNCTIONS and SETS currently), along with a bitmask of the
      // columns that can be restricted upon.
      Simba::Utils::AutoDeletePtr<Simba::ISO::CSOGUIDVector>
          pSchemaGuidVectorAuto( new Simba::ISO::CSOGUIDVector() );

      pSchemaGuidVectorAuto->push_back(
          new Simba::ISO::CSOGUID( Simba::ISO::MDSCHEMA_FUNCTIONS )
      ) ;

      pSchemaGuidVectorAuto->push_back(
          new Simba::ISO::CSOGUID( Simba::ISO::MDSCHEMA_SETS )
      ) ;
      .
      .
      .
```

Note: GetSchemas() is not called by all consumers (e.g. Excel).

Restrictions for columns are defined in bitmasks and returned as an ordered collection of unsigned long values, where each unsigned long corresponds to one of the rowsets returned.

Note that the restriction bitmask returned specifies what restrictions are supported, not what columns are restrictable. This subtle difference allows for having restrictions that do not correspond to a column within the rowset. For example, the *MEMBERS* rowset has the *tree operator* restriction that does not correspond to a column within the rowset. The restrictions are defined by Microsoft and more information is available here: http://msdn.microsoft.com/en-us/library/ms126046.aspx.

Consider a schema rowset with columns A, B, C, D, and E with ordinals 1 to 5 respectively. Now, columns A, C, and D can be restricted on and in that order. If your provider only supports restricting on columns A and D then the bitmask returned would be 0b101, denoting support for the first and third restrictions. A bitmask of 0b1001, specifying the column ordinals for column A and D, would be incorrect.

The following snippet from GetSchemas() shows the construction of the bitmasks for the MDSCHEMA_FUNCTION AND MDSCHEMA_SETS rowsets respectively.

```
Simba::ISO::CSOGUIDVector * Session::GetSchemas(
     unsigned long ** ppRestrictionSupport,      // out, own
     std::vector<std::wstring>  &  schemaNames   // out
) const
{
     .
     .
     .
     unsigned long * pRestrictions = NULL ;

     try
     {
          //Number of restrictions returned.
          const int numRestrictions = 2 ;
          pRestrictions = new unsigned long[numRestrictions] ;
          ::memset( pRestrictions, 0, numRestrictions * sizeof(
               unsigned long) );

          pRestrictions[0]=     // Restrictions for MDSCHEMA_FUNCTIONS
               (1 << (RESTRICTION_FUNCTION_IDX_LIBRARY_NAME))   |
               (1 << (RESTRICTION_FUNCTION_IDX_INTERFACE_NAME)) |
               (1 << (RESTRICTION_FUNCTION_IDX_FUNCTION_NAME))  |
               (1 << (RESTRICTION_FUNCTION_IDX_ORIGIN));

          // Restrictions for MDSCHEMA_SETS
          pRestrictions[1]=
               (1 << (RESTRICTION_SETS_IDX_CATALOG_NAME))  |
               (1 << (RESTRICTION_SETS_IDX_SCHEMA_NAME))   |
               (1 << (RESTRICTION_SETS_IDX_CUBE_NAME))     |
               (1 << (RESTRICTION_SETS_IDX_SET_NAME))      |
               (1 << (RESTRICTION_SETS_IDX_SCOPE));

          // Add the names.
          schemaNames.push_back( L"MDSCHEMA_FUNCTIONS" ) ;
          schemaNames.push_back( L"MDSCHEMA_SETS" ) ;

          // No exceptions; commit.
          *ppRestrictionSupport = pRestrictions ;
     }
     catch (...)
     {
          delete [] pRestrictions ;
          pRestrictions = NULL ;
          throw ;
     }

     // No exceptions; commit.
     return pSchemaGuidVectorAuto.Detach() ;

}
```

Returning a Schema Rowset

The next method to look at is *Session::CreateSchemaTable()* which is invoked by the calling framework when it wants to retrieve a fully-populated schema rowset of metadata. This method takes in a GUID which specifies the rowset that is to be returned, as well as a vector of restrictions (restrictions are discussed in the next section):

```
Simba::ISO::ISOTable * Session::CreateSchemaTable(
      Simba::ISO::CSOGUID   aGuid,
      const Simba::ISO::ISOCellVector  * pRestrictions
)
{
      .
      .
      .
}
```

The sample provider's implementation makes a number of calls to the DataStoreAPI to fetch the various rowsets that the caller might request. Examples of rowsets that could be requested include lists of catalogs, cubes and dimensions.

Once the requested metadata is retrieved, it is stored in an array of structures that is then wrapped by an appropriate RowSourceVector-derived object, and ultimately a TableBase object.

For example, the following snippet shows how the method handles a request for the *DBSCHEMA_CATALOGS* rowset, in which case the list of catalogs is retrieved from the DataStore API (which in turn gets it from the text provider). The metadata is then stored in a *RowSourceVector* which is embedded into an *ISOTable* object and returned to the calling framework. The calling framework will then invoke methods on this *ISOTable* object to retrieve the data:

```
Simba::ISO::ISOTable * Session::CreateSchemaTable(
      Simba::ISO::CSOGUID   aGuid,
      const Simba::ISO::ISOCellVector * pRestrictions
)
{
      .
      .

      if (IsEqualGUID( aGuid, DBSCHEMA_CATALOGS ))
      {
            CatalogRestriction restrict(*pColumnInfoVector,
                  *pRestrictions, m_sessionCatalog);
            DSCatalogMetadata * pMetadata = NULL ;
            long  numElements = 0 ;

            dsStatus =

      Simba::DataStoreAPI::DataStoreAPI::GetInstance().ListCatalogs(
                  m_SessionKey,
                  restrict.GetCatalogName(),
                restrict.GetOptionFlags(),
                &pMetadata,
                &numElements
           ) ;

           if (dsStatus != S_OK)
           {
                CustomerError err( dsStatus ) ;
           throw Simba::ISO::CSOCommonException(
                err.ErrorMessageUnicode() ) ;
           }

           // See if we found any.  If not, create an empty table.
           if ((pMetadata != NULL) && (numElements != 0))
                pRowSourceVector = new CatalogRowSourceVector(
                     pMetadata, numElements ) ;
           else
                pRowSourceVector = new RowSourceVector() ;
      }

      .
      .


      Simba::Utils::AutoReleasePtr<Simba::ISO::ISOTable>
           pISOTable( new TableBase( pColumnInfoVector.Detach(),
           pRowSourceVector.Detach()));

      CLASS_EXIT_EX_W( L"Session", L"CreateSchemaTable",
           GetSessionInfo()) ;

      // No exceptions; commit.
      return pISOTable.Detach() ;
}
```

Additional details about the *RowSourceVector* and returning data will be covered in a later section.

Although the sample implementation hard codes the metadata (since it only stores OLAP data in .csv files), you will more likely want to retrieve metadata from your own data store and cache it within your provider. If you are using Simba's MDX Engine, this metadata cache should be located in your ISM layer (see the *MDX Engine Developer's Guide* for more detail). Be aware of threading issues, since your metadata cache may be accessed from multiple threads, particularly during asynchronous MDX query execution in your ODBO provider or servicing concurrent requests in your XMLA provider.

In the sample provider, some of the schema rowsets return empty rowsets. This behavior ensures that the provider is compatible with applications that require those schemas to be present, but returning an empty rowset is also acceptable. If your data source is capable of supporting those schemas, then you can implement support for them so that actual data is returned. Otherwise, you can opt to return an empty rowset (as the sample provider does) and ignore the compatibility restrictions.

Handling Restrictions

Restrictions are values specified by the consumer that filter the returned schema rowset. This causes the provider to return only those rows containing values that match the specified restrictions.

This is accomplished in the provider by comparing the restriction values to the data stored in specific columns and returning the data for only those rows that pass the filter. For example, when *MDSCHEMA_MEMBERS* is retrieved with a restriction on *DIMENSION_UNIQUE_NAME*, then only the members that have a matching value on *DIMENSION_UNIQUE_NAME* should be returned.

When the *Session::CreateSchemaTable()* method is invoked by the calling framework, it takes in a collection (vector) of *ISOCell* objects representing the restriction values to be used.

The sample implementation uses a family of classes derived from the *Restriction* class (defined in *<YourProviderFolder>\Win32\Provider\ISOImpl\Restrictions.h*) to convert this *ISOCellVector* into a form suitable for the DataStoreAPI.

For example, when the calling framework requests the catalog schema rowset, the *CreateSchemaTable()* method constructs a *CatalogRestriction* which takes any restrictions passed in and performs the restriction logic specific to the catalog schema rowset:

```
Simba::ISO::ISOTable * Session::CreateSchemaTable(
      Simba::ISO::CSOGUID                aGuid,
      const Simba::ISO::ISOCellVector  * pRestrictions
)
{
      CLASS_ENTER_EX_W(L"Session", L"CreateSchemaTable",
            GetSessionInfo());
      CLASS_INFO_EX_W(L"Session", L"CreateSchemaTable" << NEW_LINE <<
            CTraceableSchema( aGuid ) << NEW_LINE, GetSessionInfo());

      _CheckForCatalogChange() ;

      DSStatus
      dsStatus = S_OK ;
      SchemaTableDescriptor tableDescriptor( aGuid ) ;
      Simba::Utils::AutoReleasePtr<RowSourceVector> pRowSourceVector ;

      Simba::Utils::AutoReleasePtr<Simba::ISO::ISOColumnInfoVector>
            pColumnInfoVector(tableDescriptor.CreateColumnInfo(false));

      if (IsEqualGUID( aGuid, DBSCHEMA_CATALOGS ))
      {
            CatalogRestriction restrict(*pColumnInfoVector,
                  *pRestrictions, m_sessionCatalog);
```

```
          DSCatalogMetadata * pMetadata = NULL ;
          long  numElements = 0 ;

          dsStatus =

          Simba::DataStoreAPI::DataStoreAPI::GetInstance().ListCatalogs(
               m_SessionKey,
               restrict.GetCatalogName(),
               restrict.GetOptionFlags(),
               &pMetadata,
               &numElements
          ) ;
          .
          .
```

The *CatalogRestriction* object is then used in calling the DataStoreAPI to get the filtered catalog rowset (in the call to *ListCatalogs()*).

In your implementation you will need to modify or replace the *restriction*-derived classes with your own since your restriction implementation will likely be different from the sample. For example, you could implement restrictions directly in your provider's metadata cache.

Another method which will require modification if your provider supports additional custom schema rowsets is *RestrictionMapper::_GetRestrictionMappings()*. This method is invoked when the calling framework maps restrictions to rowset columns (i.e. when filtering the columns). The method takes in the GUID for the schema rowset that is being filtered and returns an array of restriction column indices along with the number of indices.

The following code snippet shows how this method checks for the GUID of the catalog schema rowset, and then returns a hard-coded array of restriction column indices:

```
const unsigned long * RestrictionMapper::_GetRestrictionMappings(
     Simba::ISO::CSOGUID     guid,
     unsigned long         & numRestrictions
)
{
     // This helper function returns a pointer to the array of
     // constants that define the mappings between column ordinals and
     // restriction indices for the schema rowset identified by the
     // given GUID.  The size of the array is returned by reference.

     // The arrays of constants returned by this function are defined
     // in ISOConst.h.

     if (IsEqualGUID( guid, DBSCHEMA_CATALOGS ))
     {
          numRestrictions = sizeof CATALOG_RESTRICTION_MAPPINGS /
               sizeof( unsigned long ) ;
          return CATALOG_RESTRICTION_MAPPINGS ;
     }
     else if (IsEqualGUID( guid, MDSCHEMA_CUBES ))
```

```
        {
             numRestrictions = sizeof CUBE_RESTRICTION_MAPPINGS /
                  sizeof( unsigned long ) ;
             return CUBE_RESTRICTION_MAPPINGS ;
        }
        .
        .
        .
```

Therefore if your provider supports additional custom schema rowsets, you will need to add code here to check for the GUID of the custom schema rowset and return a collection of column indices for that rowset (defined in *<YourProviderFolder>\Win32\Provider\ISOImpl\ISOConst.h*):

```
const unsigned long * RestrictionMapper::_GetRestrictionMappings(
        Simba::ISO::CSOGUID guid,
        unsigned long & numRestrictions
)
{
        // This helper function returns a pointer to the array of
        // constants that define the mappings between column ordinals and
        // restriction indices for the schema rowset identified by the
        // given GUID.  The size of the array is returned by reference.

        // The arrays of constants returned by this function are defined
        // in ISOConst.h.

        if (IsEqualGUID( guid, DBSCHEMA_CATALOGS ))
        {
             numRestrictions = sizeof CATALOG_RESTRICTION_MAPPINGS /
                  sizeof( unsigned long ) ;
             return CATALOG_RESTRICTION_MAPPINGS ;
        }
        .
        .
        .
```

Returning Values from the Schema Rowsets

The previous few sections have described how to prepare the provider to return schema metadata. This section will describe how data is actually returned.

When the calling framework invokes the *Session::CreateSchemaTable()* method, this results in the creation of a fully populated schema rowset. This rowset is stored in an *ISOTable* object that is then returned to the calling framework.

Whenever the calling framework wants to get data from this *ISOTable*, it invokes the object's *GetRow()* method to return an *ISORow* object for one of the rows in the table. It will then in turn use this row object to get at the individual *cells* in the row from which it can extract the data.

To support this, the sample provider has implemented an *ISOTable* called *TableBase* that returns an implementation of *ISORow* called *RowBase*. A *TableBase* instance is created for every rowset where data and metadata need to be stored. When a *TableBase* is created, it takes in a collection of *RowSources* that are objects that contain the actual data for each row.

Note: A collection of *RowSources* is stored in a *RowSourceVector*.

Since the data stored in any given rowset has a specific set of columns, a specific *RowSource* implementation must be created on a per rowset basis and the sample provider includes a number of *RowSource* implementations.

The *CatalogRowSource* is the simplest sample implementation because it only contains one column of data: the catalog name.

The creation of a collection of these row sources can be seen in the following snippet of *Session::CreateSchemaTable()* where a *CatalogRowSourceVector* is created from the metadata retrieved from the data store:

```
Simba::ISO::ISOTable * Session::CreateSchemaTable(
      Simba::ISO::CSOGUID aGuid,
      const Simba::ISO::ISOCellVector  * pRestrictions
)
{
      .
      .
      if (IsEqualGUID( aGuid, DBSCHEMA_CATALOGS ))
      {
            CatalogRestriction restrict(*pColumnInfoVector,
                  *pRestrictions, m_sessionCatalog);

            DSCatalogMetadata * pMetadata = NULL ;
            long numElements = 0 ;

            dsStatus =

            Simba::DataStoreAPI::DataStoreAPI::GetInstance().ListCatalogs(
                  m_SessionKey,
                  restrict.GetCatalogName(),
                  restrict.GetOptionFlags(),
                  &pMetadata,
                  &numElements
            ) ;

            .
            .

            if ((pMetadata != NULL) && (numElements != 0))
                  pRowSourceVector = new
                        CatalogRowSourceVector(pMetadata, numElements);
            else
                  pRowSourceVector = new RowSourceVector() ;
      }
      .
      .

      Simba::Utils::AutoReleasePtr<Simba::ISO::ISOTable>
           pISOTable(new TableBase( pColumnInfoVector.Detach(),
           pRowSourceVector.Detach()));

      CLASS_EXIT_EX_W( L"Session", L"CreateSchemaTable",
           GetSessionInfo()) ;

      // No exceptions; commit.
      return pISOTable.Detach();

}
```

At a later point when the calling application gets an *ISORow* from the table, the table will instruct the *RowBase* to create cells for each column using the specific row source implementation (e.g. *CatalogRowSource*) for the row. The row source will in turn call its *_GetValueAtColumn()* method to get a value for each column in the row. Therefore when creating your own implementations of *RowSource* you will need to write your own logic for its *_GetValueAtColumn()* method.

This method takes in the column index to retrieve data for along with the expected data size. It then returns actual size, the raw data itself (array of bytes), and the type of data stored (for example, integer or decimal).

The following snippet shows the implementation of *CatalogRowSource::_GetValueAtColumn()* which starts by calling the base class to perform basic error checking on the parameters. It then uses a helper method called *_StringToBinary()* defined in the RowSource base class to convert the catalog name to raw data where it is then returned to the caller:

```
ROWSOURCE_STATUS CatalogRowSource::_GetValueAtColumn(
      unsigned long      colIdx,        // in
      unsigned long      dataSize,      // in
      unsigned long    & actualSize,    // out
      BYTE             * pData,         // out
      DBTYPE           & actualType     // out
) const
{
      // This method is used to return the value at the given column
      // ordinal as raw binary data.  The length of data copied into
      // the caller's buffer is returned by reference.  This will be
      // less than or equal to the given buffer size.

      ROWSOURCE_STATUS status;

      status = MetadataRowSource::_GetValueAtColumn( colIdx, dataSize,
           actualSize, pData, actualType ) ;

      if (status != ROWSOURCE_OK)
```

```
            return status ;

        switch (colIdx)
        {
        case CATALOG_COL_CATALOG_NAME:
            return RowSource::_StringToBinary( pData,
                m_catalog.catalogName, dataSize, actualSize, false ) ;

        case CATALOG_COL_DESCRIPTION:
            return RowSource::_StringToBinary( pData,
                m_catalog.description, dataSize, actualSize ) ;

        default:
            return ROWSOURCE_BADCOLUMN ;
        }
    }
```

Returning Data

Now that the provider has been setup to expose information on what it supports and the metadata it provides, it must now be setup to execute the query and return cell data as well as metadata. This is involves handling:

- MDX queries
- Returning data as either datasets or flattened rowsets

Handling Queries

A provider is responsible for taking in a command (e.g. MDX query) from the calling framework and returning a result (i.e. a dataset or rowset).

This is accomplished in the provider via an *ISOCommand* object, and more specifically its *Execute()* method. The sample provider includes an *ISOCommand* implementation called *Command* that can be used as a starting point.

Command Creation and Cleanup

The *Session* object is responsible for creating an *ISOCommand* object via its *CreateCommand()* method and returning it to the calling framework as shown in the following snippet from the sample provider:

```
Simba::ISO::ISOCommand * Session::CreateCommand() const
{
    // The following const-cast is ok since the Command object isn't
    // going to modify the session -- it will simply ref-count it.
    return new Command( const_cast<Session *>( this ) ) ;
}
```

Note: The command's constructor takes in and stores a pointer to the *Session*. This session pointer will be used by the command later on when it calls out to the MDX engine.

Once the object has performed its required logic (i.e. it has executed the command), the SDK will call the Command's *Release()* method, and the object will delete itself when there are no more references:

```
void Command::Release()
{
      long refCount = Simba::Utils::Atomic::Decrement(
           m_lReferenceCount ) ;

      if (refCount == 0)
           delete this ;
}
```

Executing Commands

The *Execute()* method is responsible for taking in a command, passing it off to the MDX engine for processing, and returning the results in either a dataset or flattened rowset.

Along with the command text, the method takes in the GUID of the language that the command is written in (either DBGUID_SQL for SQL or MDGUID_MDX for MDX), and an *ESORESULTTYPE* enumeration specifying the type of result to return (RESULTTYPE_DATASET for dataset, RESULTTYPE_TABLE for rowset, or RESULTTYPE_NONE for non-result-generating commands such as *CREATE MEMBER*).

Note: If you do not intend to have your provider support the execution of SQL queries, then your implementation needs to throw an exception if it is passed DBGUID_SQL as the sample does.

The snippet below shows the first half of the sample's implementation of *Execute()*. The method first checks the language GUID, then creates an instance of the MDX engine included in the SimbaProvider for OLAP SDK and delegates the work of executing the command to the engine:

```
Simba::ISO::ISOResultType * Command::Execute(
      Simba::ISO::CSOGUID languageGUID,
      const wchar_t * pCommandText,
      const std::vector<std::wstring>* parameterNames,
      const Simba::ISO::ISOCellVector*  parameterValues,
      Simba::ISO::ESORESULTTYPE expectedResultType
)
{
      // This executes the given MDX query and returns either a
      // dataset, a flattened dataset, or NULL, depending on the value
      // of the last parameter. The given language GUID must be
      // MDGUID_MDX.  If it is not, this method throws an exception.

      CLASS_ENTER_EX_W(L"Command",
           L"Execute( languageGUID," << pCommandText << L"," <<
           expectedResultType    << L")",
           GET_SESSION_INFO_FROM_SESSION(m_pSession)
```

```
);

//…

// Get exclusive use of the session while the command is
// executing.  This means that with the sample implementation,
// only one command can execute at a time. For parallel
// execution, the customer should implement fine-grained locking
// in their ISM implementation.
Simba::Utils::AutoLock lock( m_pSession ) ;

m_pSession->CleanupCustomAggregates();

Simba::ISO::ISOResultType * pISOResult = NULL ;

try
{
     if ((languageGUID == Simba::ISO::CSOGUID( DBGUID_SQL )) &&
          (expectedResultType == Simba::ISO::RESULTTYPE_DATASET))
     {
          throw Simba::ISO::CSOInvalidArgException(
               L"Invalid result type for SQL command specified
               in Command::Execute()."
          ) ;
     }

     if (languageGUID != Simba::ISO::CSOGUID( MDGUID_MDX ))
     {
          throw Simba::ISO::CSOUnsupportedMethodException(
               L"Unsupported language specified in
               Command::Execute()."
          ) ;
     }

     // …
     // …

     // In the interest of avoiding double-negatives, this
     // condition is made very clear.
     bool isResultExpected = (expectedResultType !=
          Simba::ISO::RESULTTYPE_NONE) ;

     // Create an instance of the MDX Engine
     Simba::Utils::AutoDeletePtr<Simba::MDXEngine::Engine>
          pEngine (new Simba::MDXEngine::Engine( m_pSession )) ;

     // Execute the MDX command
     pEngine->Execute(
          Simba::Utils::UnicodeTools::UnicodeToTString(
          pCommandText ), params, toFree ) ;
     .
     .
```

Note that *Execute()* can be invoked with RESULTTYPE_DATASET or RESULTTYPE_TABLE even though the statement does not produce a result. In either case, the implementation of this method should return NULL as long as the command executes successfully.

If a command does not execute successfully (e.g. because of an error in the command text), throw a *CSOQueryException*.

An important aspect of the *Execute()* method is that it checks the expected result specified in the *expectedResultType* parameter. If this parameter is set to RESULTTYPE_DATASET, then the provider needs to return a dataset object. If this parameter is set to RESULTTYPE_TABLE, then the provider needs to return a flattened rowset object.

The snippet below shows the latter half of the sample *Execute()* method:

```
        .
        .
        // See if there is a result.
        if (expectedResultType == Simba::ISO::RESULTTYPE_DATASET &&
             pEngine->HasDisplayBox())
        {
             Simba::Utils::AutoReleasePtr<Dataset>
             pDataset (new Dataset(m_pSession, pEngine.Detach()));

             // No exceptions; commit result.
             pISOResult = pDataset.Detach() ;
        }
        // If a table is requested, get the flattened result from
        // the Engine. An example for this case is a drillthrough
        // result.
        else if (expectedResultType == Simba::ISO::RESULTTYPE_TABLE
             && pEngine->HasFlattenedResult())
        {
             Simba::Utils::AutoReleasePtr<FlattenedTable>
                  pISOTable ( new FlattenedTable( m_pSession,
                  pEngine.Detach()));

             // No exceptions; commit result.
             pISOResult = pISOTable.Detach() ;
        }
        else
        // No result. Let the Engine expire, since it won't be
        // needed anymore.
        {
             pISOResult = NULL ;
        }
    }
    catch (Simba::ISM::ISMException * pException)
    {
        Customer::ISM::ProvideISOExceptionThrower()->
             ThrowAsISOException( pException ) ;
    }

    CLASS_EXIT_EX_W( L"Command", L"Execute returns " << pISOResult,
        GET_SESSION_INFO_FROM_SESSION(m_pSession) ) ;

    return pISOResult ;
}
```

The code checks whether the engine has returned either a display box or flattened rowset, and whether the *expectedResultType* parameter corresponds to what has been returned.

The sample provider includes two classes that implement the *ISOResultType* that the method must return:

- **Dataset**: a dataset object connected to the display box returned from the MDX engine. This class implements *ISODataSet* that in turn derives from *ISOResultType*.
- **FlattenedTable**: a rowset connected to a flattened dataset returned from the MDX engine. This class implements *ISOTable* that in turn derives from *ISOResultType*.

These classes are discussed in further detail in the next section.

Exposing Command Results

Executing a command returns either a dataset or flattened rowset set. Depending on your ISM implementation you may need to modify one or either of the implementations in the sample provider. Below are the methods of these classes that may require modification.

Datasets

The following methods of the Dataset may need to be modified.

Dataset::GetAxisInfo()

This method returns an object derived from *ISOAxisInfo* containing structural information, such as the number of axes and the number of dimensions per axis for the dataset result.

The sample implementation exists in a class called *AxisInfo*. The constructor takes in a reference to a *DisplayBox*, which is an object returned by the MDX Engine that represents the result of an MDX query. This reference is then used throughout the implementation of *AxisInfo* to fetch elements from the display box such as specific axes or axes properties, for example.

If your ISM layer differs substantially from the sample ISM implementation, then you may need to modify or the sample's *AxisInfo* class or create your own to return additional or modified information. See the *MDX Engine Developer's Guide* for more information on the *DisplayBox* object.

Dataset::CreateAxisTable()

A provider must be able to return information about all tuples on an axis for a dataset. It should handle the *DIMENSION PROPERTIES* clause in an MDX command as well.

This method responsible for returning this information is *Dataset::CreateAxisTable()* which takes in the ordinal of the axis to return information for. This method returns the information in an *ISOTable* where each row in the rowset represents a tuple. Each row is composed of a column for the tuple ordinal followed by groups of columns, where each group corresponds to a single member of the tuple. Each column in the group represents a property of the corresponding member for that group.

Each group has columns for at least five dimensional properties: MEMBER_UNIQUE_NAME, MEMBER_CAPTION, LEVEL_UNIQUE_NAME, LEVEL_NUMBER, and DISPLAY_INFO. There will be an additional column in each group for each dimension property requested in the DIMENSION PROPERTIES clause for the axis in the MDX statement. For more information on the properties see http://technet.microsoft.com/en-us/library/ms144780.aspx.

The sample implementation uses the *AxisInfo* object when constructing this table, and returns the information in a *TableBase* object. This implementation may need to be modified to work with your own ISM implementation.

For more information on the structure of an axis table, see OLE DB for OLAP Objects and Schema Rowsets/Dataset Object/Axis Rowsets in the *OLE DB for OLAP Reference* (see http://msdn.microsoft.com/en-us/library/ms725398%28v=vs.85%29.aspx).

Dataset::CreateDataSetTable()

A dataset must be able to return information about a range of cells in its result set as well as handling the *CELL PROPERTIES* clause in an MDX command.

The *CreateDataSetTable()* method in *Dataset* is the method responsible for returning this information. The method takes in the number of cell ordinals to retrieve cell data for, along with an array of cell ordinals themselves, in increasing order. It then constructs an *ISOTable* containing information about a range of cells in the dataset result. Each row contains information for a cell in the result, and each column corresponds to one of the cell's properties.

Note that the *CELL PROPERTIES* statement specified by the consumer can be appended with any number of specific properties for which to return information. The minimum set of properties supported by the sample provider are *VALUE*, *FORMATTED_VALUE*, *CELL_ORDINAL* and FORMAT_STRING. For a complete list see: http://technet.microsoft.com/en-us/library/ms145573.aspx.

If your provider supports additional properties, then you can extend the set of supported properties in your implementation by adding additional table descriptors to the *_DatasetDefaults* array in *<YourProviderFoldder>\ SampleProvider\Win32\Provider\ISOImpl\TableDescriptors.h*.

Rowsets

If the requested data type is a rowset, multidimensional results must be returned to the calling framework in a tabular form. If you are developing an ODBO provider, flattening is required in order to support Excel 2000 and earlier versions of Excel. If you are developing an XMLA provider, flattening must be done to support the *Tabular* value for the *Format* property that can be issued by an *Execute* command. For XMLA, Microsoft Reporting Services and Tableau will execute commands and require the results in tabular form.

The process of creating this tabular result format involves “flattening” the table, and is handled by the sample provider’s MDX engine, which includes a flattening algorithm.

To ensure that the lifetime of the result set is properly managed, the sample provider includes the *FlattenedTable* class. This *ISOResultType* implementation wraps all of the calls to the MDX engine to get and work with a flattened rowset. A *FlattenedTable* class is returned by the *Command::Execute()* method when a command requests data to be returned in a flattened rowset. The calling framework will then make subsequent calls on this *IOSResultType* to get data from it.

The *FlattenedTable* class will be sufficient for most providers, especially those that use the Simba MDX engine, and will not likely need to be modified.

For more information about the flattening algorithm, see Microsoft’s *OLE DB for OLAP Reference* (see Rowsets in OLE DB for OLAP/Flattening a Dataset to Produce a Rowset for more detail).

Logging Messages

The SDK includes a logging system that allows you to add log and trace messages to your provider’s code through a series of macros on both the Windows and Linux platforms. For example, the following snippet from the *Command::Execute()* method shows the usage of the *CLASS_ENTER_EX_W* macro to log that the method has entered execution:

```
Simba::ISO::ISOResultType * Command::Execute(
      Simba::ISO::CSOGUID languageGUID,
      const wchar_t * pCommandText,
      Simba::ISO::ESORESULTTYPE expectedResultType
)
{
      CLASS_ENTER_EX_W(L"Command",
            L"Execute( languageGUID," << pCommandText << L"," <<
                  expectedResultType << L")",
            GET_SESSION_INFO_FROM_SESSION(m_pSession)
      );
      .
      .
```

The following sub sections provide more detail on the logging system and how to use these macros.

Overview of Logging System Elements

The main elements of the logging system are:

- **Logging Levels**: the SDK defines the ordered series of logging levels shown below. These are ordered by priority from top to bottom and are used to filter messages:
    - ERROR: used for logging the error messages
    - WARNING: used for logging the warning messages which are potentially harmful situations
    - LIGHT: used for logging the informational messages at a coarse-grained level
    - DETAILED: used for logging the informational messages that are more detailed
    - ENTER: used for marking the entrances to a methods
    - EXIT: used for marking the exits from methods
    - INFO: used for lowest priority informational messages
    - NONE: no logging is to be performed.
- **LogWriters**: components which write log messages to a specific target (e.g. to disk). Currently the only available target is a file.
- **Subsystems**: a hierarchy of sub configurations allowing you to specify the log writer, logging level, and enabled/disabled state for logging messages tagged with that subsystem name. A subsystem is defined using a macro (e.g. *#define SUBSYS "ISOImpl")*. If no subsystem is defined, “ROOT” is the default subsystem.

Every message that is sent for logging is tagged with a logging level that should be used when logging that message. If the logging level for that message is equal or higher in priority than the logging level defined for that subsystem, the message will be logged, otherwise it will be ignored.

The *CLASS_ENTER_EX_W* macro used in the example above, specifies a logging level of *Enter* and that the class name to be logged is “Command”. The subsystem used is “ISOImpl” (defined in *TraceISOImpl.h*). As long as the “ISOImpl” subsystem has been configured with a logging level of *ENTER* or lower, the message will be logged using whichever *LogWriter* that subsystem configuration is using.

Logging Exceptions

The logging system is designed to automatically log internal exceptions. If an exception occurs in the logging system itself, the system sends a message to the Windows event viewer. If the exception is a fatal one, then future logging requests are ignored.

Configuring the Logging System

The logging system configuration is data-driven and read by the provider at startup. The configuration settings consist of the following:

- **General “Trace Settings”**: includes:

    - **Enabled**: set to 1 to enable logging, or 0 to disable.

- **LogWriter Configurations**: contains a collection settings for each *LogWriter*. For the current *LogFile* writer the following options are available:
    - **FileName**: the base name of the log files that will be created. The value of this field must not have any path or extensions. The current date and time will be appended to the FileName value for each log file created.
    - **MaxSize**: the maximum size limit for each log file that will be created. Whenever the size of a log file exceeds this value, a new file will be created and the rest of the messages will be written to the new file.
    - **Path**: the path to the folder that log files will be created in, like "C:\logs\".
    - **Type**: Always set to "FILE" for LogFile writers.

- **Hierarchy of subsystem configuration settings**: each subsystem configuration exists under a subsystem called *ROOT*. The settings for the *ROOT* subsystem define the default configuration options used for any subsystems that do not have any options defined. The *ROOT* subsystem, and (optionally) each child subsystem has the following settings:
    - **Enabled**: set to 1 to enable logging for the subsystem, or 0 to disable.
    - **Options**: specifies one or more *LogWriters* and logging levels to use for that subsystem. This field is in the form of: <LOGWRITER>:<LOGLEVEL>;<LOGWRITER2>:<LOGLEVEL2>;... For example, a value of LogFile:INFO specifies that the *LogFile* LogWriter is to be used, and that only messages tagged with a level of INFO or higher priority are to be logged to file.

Configuring Logging on Windows

On Windows, the logging configuration information is stored in the registry under the following nodes for the sample provider:

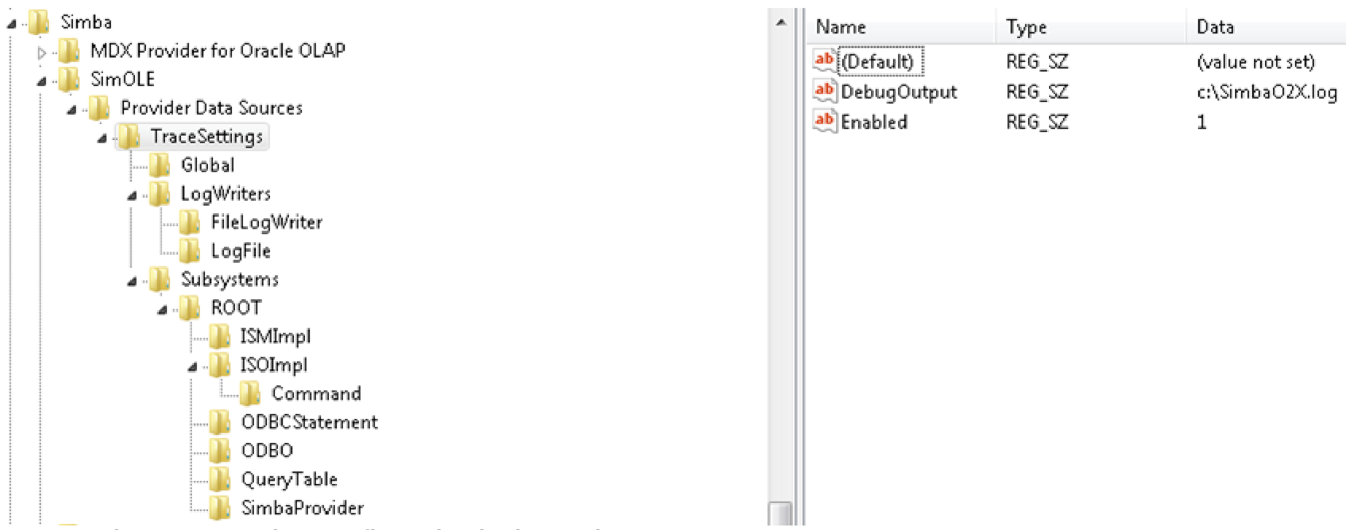
Windows 32-bit Native or Windows 64-bit:

[HKEY_LOCAL_MACHINE\SOFTWARE\Simba\SimOLE\Provider Data Sources\TraceSettings]

Windows 32-bit on Windows 64-bit:

[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Simba\SimOLE\Provider Data Sources\TraceSettings]

The following screenshot shows a sample configuration in the registry:



```
Simba
  MDX Provider for Oracle OLAP
  SimOLE
    Provider Data Sources
      TraceSettings
        Global
        LogWriters
          FileLogWriter
          LogFile
        Subsystems
          ROOT
            ISMImpl
            ISOImpl
              Command
            ODBCStatement
            ODBO
            QueryTable
            SimbaProvider
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
DebugOutput	REG_SZ	c:\SimbaO2X.log
Enabled	REG_SZ	1

Figure 2 - Logging Configuration in the Registry

At the base level are the *TraceSettings* where logging can be enabled and disabled. The *LogWriters* folder contains the settings for each *LogWriter*, and the *Subsystems* folder contains the *ROOT* folder under which each subsystem configuration is defined.

If you want to change the registry key location of the log files, then modify *<YourProviderFolder>\Win32\Provider\Main\Main.cpp* (for ODBO) or *<YourProviderFolder>\Common\JniIso\Customer.Olap.Jni.Iso\NativeIsoClassFactory.cpp* (for Java XMLA) and change the value of *TRACING_ROOT_KEY_PATH* to your registry key location. For .NET XMLA, modify *<YourProviderFolder>\dotNET\ISO.NET\Customer.Olap.ManagedIso/Provider.cpp* and set the registry key location in the constructor.

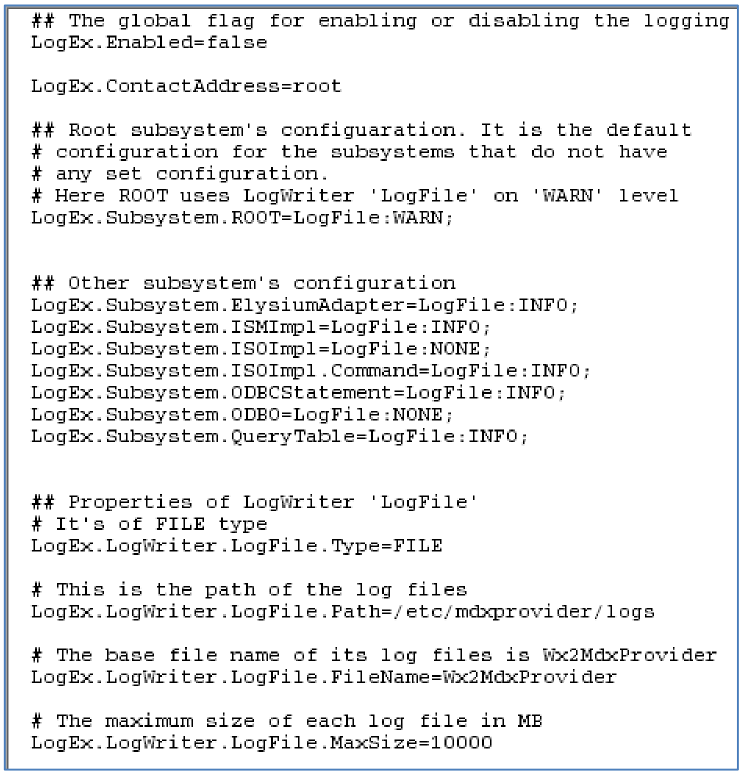
The SDK includes the following sample registry files in *<YourProviderFolder>\Win32\Provider\LogImpl\Configuration\WindowsConfiguration* to help you quickly create these settings:

- **32on64 OLAP Connector Trace Settings ON**: creates and enables the settings for a 32-bit provider running on 64-bit Windows.
- **32on64 OLAP Connector Trace Settings OFF**: creates and disables the settings for a 32-bit provider running on 64-bit Windows.
- **Native OLAP Connector Trace Settings ON**: creates and enables the settings for a provider running on the same bitness of Windows.
- **Native OLAP Connector Trace Settings OFF**: creates and enables the settings for a provider running on the same bitness of Windows.

If you modified the registry location, then edit the sample registry files to match your new registry location.

Configuring Logging on Linux

On Linux the sample logging configuration is defined in "*tracesettings.conf*" file located in *<YourProviderFolder>\Win32\Provider\LogImpl\Configuration\LinuxConfiguration.*



```
## The global flag for enabling or disabling the logging
LogEx.Enabled=false

LogEx.ContactAddress=root

## Root subsystem's configuaration. It is the default
# configuration for the subsystems that do not have
# any set configuration.
# Here ROOT uses LogWriter 'LogFile' on 'WARN' level
LogEx.Subsystem.ROOT=LogFile:WARN;


## Other subsystem's configuration
LogEx.Subsystem.ElysiumAdapter=LogFile:INFO;
LogEx.Subsystem.ISMImpl=LogFile:INFO;
LogEx.Subsystem.ISOImpl=LogFile:NONE;
LogEx.Subsystem.ISOImpl.Command=LogFile:INFO;
LogEx.Subsystem.ODBCStatement=LogFile:INFO;
LogEx.Subsystem.ODBO=LogFile:NONE;
LogEx.Subsystem.QueryTable=LogFile:INFO;


## Properties of LogWriter 'LogFile'
# It's of FILE type
LogEx.LogWriter.LogFile.Type=FILE

# This is the path of the log files
LogEx.LogWriter.LogFile.Path=/etc/mdxprovider/logs

# The base file name of its log files is Wx2MdxProvider
LogEx.LogWriter.LogFile.FileName=Wx2MdxProvider

# The maximum size of each log file in MB
LogEx.LogWriter.LogFile.MaxSize=10000
```

Figure 3 - Screenshot of Linux configuration file

The included ant script for building Java XMLA will package *tracesettings.conf* within the WAR file at *WEB-INF/classes/com/simba/*. The encoding of this file must be set to UTF-8.

The following is the list of properties you can set in this configuration file:

- **LogEx.Enabled={true, false}**: enables/disables the logging subsystem.
- **LogEx.ContactAddress={email, unix username}**: deprecated. This property was used only when an internal exception occurred in the logging system. It specifies the address of the contact person to send information to when a logging system exception occurred. One or more email addresses and/or UNIX names can be specified in a comma-separated list.
- **LogEx.Subsystem.ROOT={OPTIONS}**: the options related to the *ROOT* subsystem.
- **LogEx.Subsystem.X.Y={OPTIONS}**: the options related to the X.Y subsystem. You can have as many levels in your hierarchy as you want.
- **LogEx.Subsystem.X.Y.Enabled={true,false}**: determines whether a subsystem should be enabled or disabled. Alternatively, you can set the log level of all of the LogWriters of this subsystem to NONE.
- **LogEx.LogWriter.X.Type=FILE**: introduces a new LogWriter by the name of X and type of FILE.
- **LogEx.LogWriter.X.Path**: sets the path of LogWriter X.

- **LogEx.LogWriter.X.FileName**: sets the filename of LogWriter X.
- **LogEx.LogWriter.X.MaxSize**: sets the maximum size limit of LogWriter X.

Macros for Logging

The logging system macros are defined in *Simba\Common\Provider\LogEx\LogExMacros.h* and are grouped by the type of functionality they provide.

Macros generally hardcode the logging level that is to be used, and take in the following parameters:

- **Class name**: the name of the class under a subsystem to be logged with.
- **Arguments**: the message to display.
- **Data**: additional data to pass in with message. This can be set to null or to an object that implements *ILoggable* (discussed below in more detail). This is only necessary if the session ID or user name are to be logged. Otherwise, this can be set to null.

In the following snippet, *CLASS_ENTER_EX_W* is being used;

```
CLASS_ENTER_EX_W(L"Command",
      L"Execute( languageGUID," << pCommandText << L"," <<
           expectedResultType    << L")",
      GET_SESSION_INFO_FROM_SESSION(m_pSession)
);
```

The definition for this macro in *LogExMacros.h* is as follows:

```
CLASS_EX_W(zSubSys, Simba::LogEx::LoggingLevelUtils::logLevelEnter,
args, data)
```

Here we can see that the macro hard codes the *Enter* level so the message will only be filtered out if the subsystem that was specified has been configured with a higher priority logging level.

ILoggable Objects

Additional information can be passed in to the data parameter of most logging macros, by specifying an object that implements *ILoggable*. The *ILoggable* interface provides the user name and session ID to the log output. In future releases the *ToString()* method may be supported to append additional string information to the logged message.

Appendix A – ISO Interfaces and Classes

The following table lists the ISO interfaces and classes. The "CSO" classes are helper classes that you can use to implement the ISO interfaces.

Table 1 – ISO Interfaces and Classes

Interface/Class Name	Purpose
ISOClassFactory	A factory for *ISODatasource* objects. Also provides information about the provider.
ISODatasource	Represents a connection to a data source. Also provides a factory for creating *ISOSession* objects as well as information about the data source.
ISOSession	*ISOSession* represents a context for session-specific statements such as the MDX "CREATE MEMBER" statement. It also provides a factory for creating *ISOCommand* objects as well as a mechanism for retrieving the OLAP metadata from the underlying database.
ISOCommand	*ISOCommand* provides a factory for creating *ISODataset* and *ISOTable* objects by executing queries.
ISODataset	This is an interface to an object that contains the result of an MDX Query.
ISOTable	An *ISOTable* is a table. It contains column information and is a factory for *ISORows*.
ISOResultType	*ISOTable* and *ISODataset* inherit from this interface. This interface is used to determine whether the result of a command execution is an *ISOTable* or an *ISODataset*.
ISOAxisInfo	This interface is used to describe the structure of a 'cross-tab' that results from an MDX query.
ISOColumnInfo	This interface is used to describe a column in a table.

ISOColumnInfoVector	This interface represents a vector containing *ISOColumnInfo* objects.
ISOColumnInfoFactory	This interface provides a mechanism for client code to create *ISOColumnInfo's* and *ISOColumnInfoVectors* without being aware of the underlying concrete subclasses involved.
ISOCell	An *ISOCell* represents data. Each *ISOCell* represents a data value in a table. Multiple *ISOCells* make up a row.
ISOCellVector	This interface represents a collection of *ISOCells* as a vector.
ISOCellFactory	This interface provides a mechanism for client code to create *ISOCells* and *ISOCellVectors* without being aware of the underlying concrete subclasses involved.
ISOMessageDescFactory	A factory for localized error messages used by *CSOException* and its sub-classes.
ISORow	An *ISORow* is read from an *ISOTable*. An *ISORow* contains *ISOCells* and has access to the *ISOColumnInfoVector* for the table.
CSOConnectionInfo	Contains information needed to connect to the database, such as user name, password and provider string.
CSOInfo	Contains the information returned by *ISODatasource::GetInfo*.
CSOProviderInfo	Contains the information returned by *ISOClassFactory::GetProviderInfo*.
CSOLiteral	Contains the information returned by *ISODatasource::GetLiteralInfo*.

CSOException	A standard exception class, intended to be used in place of std::exception (this has the additional feature of wide character support). *CSOException* is the base class for all exception classes which follow.
CSOCommonException	HRESULT: E_FAIL. Thrown in circumstances where a customer specific error has occurred
CSOQueryException	HRESULT: DB_E_ERRORSINCOMMAND. Thrown when an error occurs while executing a query. This represents an error triggered by a mistake in the command text.
CSOUnsupportedMethodException	HRESULT: E_NOTIMPL.Thrown when an unsupported optional method is called.
CSOEndOfTableException	HRESULT: DB_S_ENDOFROWSET. Thrown when an end of table is reached during a call to *ISOTable::GotoNextRow* or *ISOTable::GotoPreviousRow*. This is not an error condition.
CSOAuthFailedException	HRESULT: DB_SEC_E_AUTH_FAILED. Thrown when authentication fails for the user information returned by the *GetUserID* and *GetPassword* methods.
CSOCanceledException	HRESULT: DB_E_CANCELED. Thrown when the user presses cancel on the dialog box when logging on.
CSOInvalidArgException	HRESULT: E_INVALIDARG. Thrown when an argument in a method is invalid. For example: when a value is NULL or out of a valid range.

Note: All of the ISO interfaces and classes can be found in *Simba/Common/Provider/ISO* in the *ISODatabase.h* and *ISOMessageDescFactory.h* files. Please refer to these files for complete details including method declarations.

Appendix B – Supported API Interfaces

ODBO Supported Interfaces and Methods

Simba provides all the OLE DB COM objects required to implement an OLE DB for OLAP provider, including Datasource, Session, Command, Rowset and the multidimensional data-specific Dataset object. We also support ODBO-specific interfaces, including *IMDDataset* and *IMDRangeRowset*.

For each of these objects, we support all the mandatory interfaces and most of the optional interfaces to ensure the flexibility and maximum interoperability of your OLAP provider.

For more information on these interfaces, see the *OLE DB Programmer's Reference* (Microsoft). You can view this at the Microsoft Developer Network web site at http://msdn.microsoft.com/library.

XMLA Supported Schema Rowsets and Properties

The following table lists the properties supported by the XMLA layer.

Table 2 - Property Support

Supported Properties	AxisFormat
	BeginRange
	Catalog
	Content
	DataSourceInfo
	EndRange
	Format
	LocaleIdentifier
	MDXSupport

	Password
	ProviderName
	ProviderVersion
	StateSupport
	UserName

Properties not Supported	Cube
	Timeout
	ClientProcessID
	DbpropMsmdActivityID
	DbpropMsmdCurrentActivityID
	DbpropMsmdFlattened2
	DbpropMsmdMDXCompatibility
	DbpropMsmdOptimizeResponse
	DbpropMsmdSubqueries
	Dialect
	MdxMissingMemberMode
	ReturnCellProperties
	SafetyOptions
	ShowHiddenCubes
	SspropInitAppName

Note: The Simba XMLA layers recognize and accepts the unsupported properties. However, if a consumer application supplies them no action is taken.

The following table lists all available schema rowsets. Note that the first four rowsets are implemented by Simba. The remaining schema rowsets are part of your ISO implementation and, except for *DBSCHEMA_CATALOGS*, are also required for ODBO providers. You can implement additional schema rowsets not listed here in your ISO implementation. Consumer applications will be able to retrieve the data in your additional schema rowsets via the *Discover* method.

Table 3 - Schema Rowset Support

Schema Rowset	Comments
DISCOVER_DATASOURCES	ProviderType restriction is not supported.
DISCOVER_PROPERTIES	
DISCOVER_ENUMERATORS	
DISCOVER_SCHEMA_ROWSETS	
DISCOVER_KEYWORDS	
DISCOVER_LITERALS	Additional informational column, LiteralID, at the end of each row in the rowset.
DBSCHEMA_CATALOGS	
MDSCHEMA_CUBES	
MDSCHEMA_DIMENSIONS	
MDSCHEMA_HIERARCHIES	
MDSCHEMA_LEVELS	
MDSCHEMA_MEASURES	
MDSCHEMA_MEMBERS	
MDSCHEMA_PROPERTIES	

MDSCHEMA_FUNCTIONS	
MDSCHEMA_KPIS	
MDSCHEMA_SETS	
MDSCHEMA_MEASUREGROUPS	
MDSCHEMA_MEASUREGROUP_DIMENSIONS	

Appendix C – Deployment and Configuration

C++ XMLA Configuration

There are a number of configurable items for monitoring and tuning the operation of your C++ XMLA provider. The configurable items for your provider include:

- Dataset XSD schema
- Data sources
- Connection configuration including sessions and pooling
- ASP.NET settings
- Performance counters

Dataset XSD Schema

You can customize the XSD schema for your XMLA provider that the *Execute* method sends back with each dataset result. The *DatasetSchema.xml* file is located at *<YourProviderFolder>/dotNET/XMLA VS2015* and *<YourProviderFolder>/Java|Eclipse Workspace|TestFiles*. You may want to modify this file to reflect the datasets returned by your provider more accurately. For example, remove member or cell property information not returned by your XMLA provider or add additional cell or member property information that your XMLA provider returns that is not in the existing XSD schema.

If you decide to move the location of the *DatasetSchema.xml* file then you will need to modify the DatasetSchemaPath key within the appSettings section in the *Web.config* located at *<YourProviderFolder>/dotNET/XMLA*. You cannot rename this file as the XMLA layer requires that the dataset XSD schema be located in the *DatasetSchema.xml*. For Java XMLA, the location is set in the *xmla.properties* file located at *Simba/Java/src/com/simba*.

Datasource Configuration

The *DataSources.xml* contains a list of data sources that your XMLA provider can connect to. For example, your XMLA provider may be located on a web server and you could have multiple data sources on separate machines to which it could connect. To insert a new data source, copy the existing <DataSource> section and update the element values to match the new data source. The <URL> element is the URL to your XMLA provider. The <DataSourceInfo> element is the connection string to use to connect to the new data source. For more information on the connection string, see "Connection Configuration" on page 61.

Note: A data source entry can be made to an XMLA provider that is located on another machine. By setting the <URL> element to the remote machine, consumer applications can find and connect to the XMLA provider located on remote machines.

For a request to succeed, the consumer application must supply a *DataSourceInfo* property value that matches at least one *<DataSourceInfo>* element. If the consumer application does not supply a *DataSourceInfo* property value, then the XMLA provider will try to connect to the first *<DataSource>* defined in the file.

Note: You may have identical *<DataSourceInfo>* elements in your *DataSources.xml* file if the URLs used to connect to those data sources are different.

If you decide to move the location of the *DataSources.xml* file then you will need to modify the *DatasourcesPath* key within the *appSettings* section in the Web.config located at *<YourProviderFolder>/dotNET/XMLA*. You cannot rename this file, as the XMLA layer requires that the data source information be located in the *DataSources.xml*. For Java XMLA, the location is set in the *xmla.properties* file located at *Simba/Java/src/com/simba*.

Connection Configuration

The ISO.NET layer includes a connection pooling system to help improve the scalability of your XMLA provider. The basic idea behind connection pooling is to avoid opening a new connection to the data source every time one is requested. Instead, pools of connections are maintained, and open connections are taken from these pools when needed. This also provides the opportunity to limit the maximum number of connections that the provider can have open at any given time.

By default, connection pooling is enabled for any data sources listed in *DataSources.xml*. To disable connection pooling for a data source, add *Pooling=false* to the *<DataSourceInfo>* element for a data source as listed in the example below:

```
<DataSourceInfo>Data Source=localhost;Pooling=false</DataSourceInfo>
```

Connection pooling requires that all connections in a pool be compatible. That is, they are for the same data source and catalog and have the same security context.

Connection strings are used to determine how connections are pooled. All connections within a pool have identical connection strings. As a result, a new pool is created if the connection string for a new request does not match any existing pool. A connection string is constructed for a data source using the value from the *<DataSourceInfo>* element for the data source in the *DataSources.xml* file and adding the values for the *Catalog*, *Password*, and *UserName* properties, if they were specified in the request made by an XMLA consumer application.

The following is a complete list of properties that you can specify in the *<DataSourceInfo>* element. These become part of the connection string used by the connection pooling system and correspond to the properties in the CSOConnectionInfo object sent to your ISO layer.

- **Datasource** The name of the data source to connect to.
- **Initial Catalog** The name of the catalog to use when the connection is first opened.
- **User ID** The name of the user or role to connect as.
- **Password** The password corresponding to the user ID.
- **Extended Properties** A value containing another connection string for provider-specific items. This is a "connection string within a connection string".

Note: If you specify values for the "Initial Catalog", "User ID", or "Password" properties in the *<DataSourceInfo>* element, they will be ignored by the XMLA layer and the connection pooling system. Values for the corresponding XMLA properties must be specified in an XMLA request in order to have any effect. This behavior may change in a future version of the SDK.

ASP.NET Settings

Long running requests may not complete properly on older machines with a Pentium III or earlier processor or under 512 MB RAM. This could be caused by retrieving a large schema rowset or executing a complex MDX query. When using IIS 5, the root configuration file for ASP.NET settings is located at *<WindowsDirectory>\Microsoft.NET\Framework\<Version>\CONFIG\Machine.config*.

The *<processModel>* section contains the ASP.NET settings. If requests are failing because of memory limitations then increase the *memoryLimit* setting. If requests are failing because queries are taking too long to execute then increase the *responseDeadlockInterval* value. When using IIS 6, the configuration settings described above are set using the IIS administrative UI.

Performance Counters

There are three performance objects you can use to monitor the performance of your provider. The Simba Olap Connection Pooling object monitors the performance of the connection pooling system. The Simba XMLA: Sessions object provides information about XMLA sessions. Finally, the Simba XMLA: Web Service object provides information about the requests that your provider receives.

To use these performance objects and for detailed information on each performance counter start the *Performance* tool under *Control Panel | Administrative Tools*. Click the [+] button on the toolbar and select the appropriate item in the *Performance object* drop-down list. Each counter has its own help description that can be viewed by first selecting the counter, then clicking the Explain button.

To enable the performance counters set the value of *PerformanceTrackingEnabled* key within the *appSettings* section in the *Web.config* file to *true*. To control how often the performance counters sample data modify the value of the *PerformanceDataCollectionInterval* key within the *appSettings* section.

To install the performance counters when deploying your XMLA provider you need to run *InstallUtil.exe* on the *Simba.Xmla.dll* and *Simba.Olap.Common.dll* assemblies.

Appendix D – XMLA Authentication Options

Introduction

Session authentication and communication security are distinct concepts that work hand in hand to provide a secure system. (To clarify the distinction between the two, consider the intranet environment. In this case, communication security is probably not as important as user authentication and access control is.)

This appendix focuses on the options available for authenticating and securing your XMLA provider.

Session Authentication and Access Control Methods

SimbaProvider SDK implements XMLA using SOAP over HTTP. Figure 4 is a simplified diagram showing the data-flow through a SimbaProvider-based XMLA provider, starting from the XMLA client and ending at your data source.

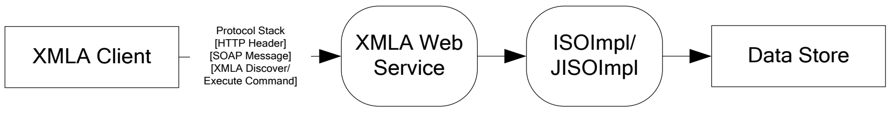


Figure 4 - XMLA Data Flow Diagram

The relevant communication protocols are:

- HTTP
- SOAP
- XML For Analysis (XMLA)

The following XMLA request demonstrates the usage and bindings of the above protocols:

```
POST /XmlaWebService/ HTTP/1.1
Accept: */*
Accept-Language: en-ca
soapaction: "urn:schemas-microsoft-com:xml-analysis:Discover"
Content-Type: text/xml
UA-CPU: x86
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
.NET CLR 1.1.4322; .NET CLR 2.0.50727; .NET CLR 3.0.04506.30;
.NET CLR 3.0.04506.648)
Host: localhost:81
Content-Length: 576
Connection: Keep-Alive
Cache-Control: no-cache
Cookie: JSESSIONID=B8F0B88E34FC1C90893494F9C4CAB106
```

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi = "http://www.w3.org/2001/XMLSchemainstance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
     <Discover xmlns="urn:schemas-microsoft-com:xml-analysis"
     SOAP-ENV:encodingStyle="http://
     schemas.xmlsoap.org/soap/encoding/">
          <RequestType>DISCOVER_DATASOURCES</RequestType>
          <Restrictions>
               <RestrictionList>
               </RestrictionList>
          </Restrictions>
          <Properties>
          <PropertyList>
               <Format>Tabular</Format>
          </PropertyList></Properties>
     </Discover>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

XMLA is built on SOAP typically running over HTTP. The portion of the above highlighted in blue is the HTTP header. The SOAP envelope is highlighted in brown. The SOAP envelope wraps the XMLA message. Finally, the XMLA message is highlighted in red.

Within the OSI model, these three protocols all fall within layer 7 (application). HTTP and SOAP are not apparent to an end user as they are binding protocols that convey requests and responses between the XMLA application and the XMLA server. Using the SimbaProvider SDK, your responsibility within the ISO/JISO layer is in handling the XMLA request; the other protocols are handled by the appropriate external frameworks (.NET/IIS or J2EE/JBoss).

You have the option of implementing authentication at each protocol involved herein (i.e., HTTP, SOAP and XMLA). The options are:

- **HTTP authentication**: authentication via passing HTTP basic, or digest, authentication header.
- **WS-Security**: authentication via passing user name and password in WSSecurity SOAP header.
- **XMLA Request Properties**: authentication via XMLA user name and password request properties.

These are not mutually exclusive; you will likely need to use at least one but probably not all three. Within SimbaProvider SDK, you cannot use WS-Security header together with XMLA Request Properties as the former will override the latter.

The following diagram demonstrates how the protocol stack is handled in case of a Simba Provider-based XMLA provider:

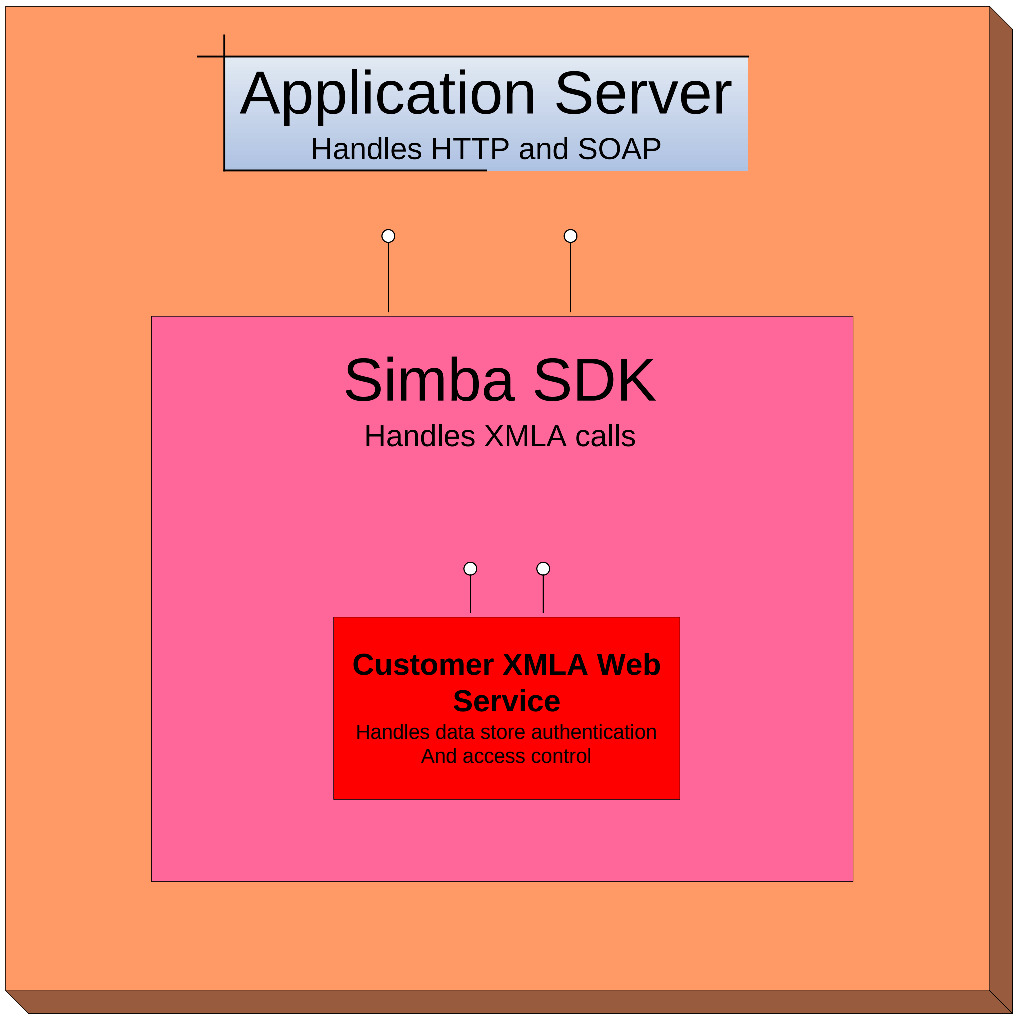


Figure 5 - XMLA Protocol Stack Handling

The application server handles the HTTP call. (For .NET XMLA, the application server is Microsoft IIS; for Java XMLA, the server is JBoss or Tomcat.) HTTP authentication is primarily handled by the application server. The SimbaProvider SDK components handling the XMLA calls—either the XMLA layer or the JXMLA layer—operates on the HTTP authentication headers which the application server supplies.

The WS-Security SOAP header is parsed by the application server using the SOAP framework that is part of the application server services. The WS-Security header is available to your XMLA web service instance through the XMLA Property Manager class (Simba.Xmla.PropertiesManager member of Simba.Xmla.XmlaWebService class). SimbaProvider currently only supports the username and password headers. A full-fledged implementation of WSSecurity may involve additional elements such as X509 certificates and digital signatures. (Please contact Simba if you are interested in using any of these advanced elements.)

In most cases, you will not need to access the Property Manager directly. In .NET XMLA, these properties are propagated to your data source through the *CSOConnectionInfo* class. In Java XMLA these properties are passed to your data source in the same fashion.

Only when you are developing a pure Java JISO implementation will you need to access the properties directly.

When the *WS-Security* header is absent, the Property Manager will be populated with the username and password properties that were submitted as part of the XMLA Request Properties.

If credentials are supplied in both the WS-Security header and the XMLA properties, the WS-Security header supersedes the XMLA properties.

XMLA Request Properties

The most direct approach for authentication is to have the username and password supplied as part of the XMLA request properties as shown below:

```
<Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://
schemas.xmlsoap.org/soap/envelope/">
     <Header>
          <BeginSession mustUnderstand="1"
               xmlns="urn:schemasmicrosoftcom:xml-analysis" />
     </Header>
     <Body>
          <Execute xmlns="urn:schemas-microsoft-com:xml-analysis">
          <Command>
               <Statement />
          </Command>
          <Properties>
               <PropertyList>
                    <LocaleIdentifier>4105</LocaleIdentifier>
                    <DataSourceInfo>Local Simba Test Server
                    </DataSourceInfo>
                    <Catalog>Adventure Works DW</Catalog>
                    <UserName>Fred</UserName>
                    <Password>XXX</Password>
               </PropertyList>
          </Properties>
     </Execute>
     </Body>
</Envelope>
```

The properties are passed onto the CSOConnectionInfo/ConnectionStringMap object that is used to create a connection to your data source object. You will be able to authenticate the end-user using the credential in the PropertiesList element.

The drawback to this approach is that these credentials must be passed in by the XMLA client. If the client does not include or is otherwise unable to use XMLA properties then you will not be able to use this method.

WS-Security

SimbaProvider implements a simplified version of WS-Security that only recognizes the Security SOAP header containing a UsernameToken. (Please contact Simba if you are interested in using any advanced elements.) The UsernameToken SOAP header has the following format:

```
<Envelope xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd=http://www.w3.org/2001/XMLSchema xmlns="http://
schemas.xmlsoap.org/soap/envelope/">
<Header>
      <Security>
            <UsernameToken>
                  <Username>Bob</Username>
                  <Password Type="PasswordText”>Foo</Password>
            </UsernameToken>
      </Security>
      </Header>
</Envelope>
```

The username and password specified in the SOAP header are passed onto the CSOConnectionInfo/ConnectionStringMap object and used to create a connection to your data source object.

In detail, the WS-Security header is parsed and the username and password in the header is used to populate the *SecurityHeader* field of the *XmlaWebService* class.

The SecurityHeader field is an instance of the Security class while the XmlaWebService class (which all SimbaProvider SDK XMLA providers inherit from) implements an XMLA Web Service. The Security class represents the WSSecurity header and contains a field UsernameToken. The code for the Security and UsernameTokenElement is listed below:

```
[In C#]
public class Security : SoapHeader
{
      public UsernameTokenElement UsernameToken ;
      public Security()
      {
            this.UsernameToken = null ;
      }
}

public class UsernameTokenElement
{
      /// <summary>
      /// Contains the user name of the authenticating party sent by
      /// the client.
      /// </summary>
      public string Username ;
      /// <summary>
      /// Contains the plaintext password of the authenticating party
      /// sent by the client.
      /// </summary>
```

```
        /// <remarks>
        /// The XMLA SDK currently only supports plaintext passwords,
        /// and does not support the <c>Type</c> attribute on the
        /// &lt;Password&gt; element.
        /// </remarks>
        public string Password ;
        /// <summary>
        /// Default constructor required for serialization.
        /// </summary>
        public UsernameTokenElement()
        {
             this.Username = "" ;
             this.Password = "" ;
        }
   } // end class UsernameTokenElement

   [In Java]
   public class Security
   {
        public UsernameTokenElement UsernameToken;
        public Security()
        {
             this.UsernameToken = null;
        }
   }

   public class UsernameTokenElement
   {
        /**
        * Contains the user name of the authenticating party sent by
        * the client.
        */
        public String Username;
        /**
        * Contains the plaintext password of the authenticating party
        * sent by the client.
        *
        * The XMLA SDK currently only supports plaintext passwords, and
        * does not support the
        * <code>Type</code> attribute on the &lt;Password&gt; element.
        * /
        public String Password;
        /**
        * Default constructor required for serialization.
        */
        public UsernameTokenElement()
        {
             this.Username = "";
             this.Password = "";
        }
   }
```

The WS-Security header must be supplied by the XMLA client in order for you to be able to use this authentication method.

HTTP Authentication

Various forms of the authentication through HTTP header are ubiquitously implemented among client applications. However, the same cannot be said for WS-Security header or XMLA Request Properties. Therefore, it is very likely that you can take advantage of some form of HTTP authentication.

There are two methods of HTTP authentication:

- HTTP basic access authentication: in this method the user name and password are Base64 encoded and sent as part of the HTTP header.
- HTTP digest access authentication: this is similar to the above except a hash of the password (appended with some other data) is sent over the wire. The server must be aware of the user's password to be able to authenticate the user.

With either method, the user name and password can be obtained from the application server in the XMLA web service class. However, the .NET XMLA SDK does not provide a mechanism to pass the user name and password to your ISO implementation. That is, your instance of *CSOConnectionInfo/ConnectionStringMap* is not populated.

The best way for you to pass the user credentials to your ISO implementation is to use the username and password fields of *SecurityHeader.UsernameToken* object as explained in the WS-Security section above.

For HTTP basic authentication, Java XMLA will pass the user and password to your ISO implementation within *CSOConnectionInfo*.

Windows Integrated Authentication

MSDN defines Windows Integrated Authentication (WIA) as:

"A negotiated, single sign on type of authentication that is the Windows implementation of Simple and Protected GSSAPI Negotiation Mechanism (SPNEGO). SPNEGO negotiation determines whether authentication is handled by Kerberos or NTLM. Kerberos is the preferred mechanism. Negotiate authentication on Windows-based systems is also called Windows Integrated Authentication."

Windows Integrated Security is a term associated with SSPI (Security Support Provider Interface) functionality that was introduced with Microsoft Windows 2000. A number of protocols can operate under the auspice of WIA including Kerberos and NTLM. WIA does not refer to a standard or an authentication protocol per se but if WIA is selected as an option for authenticating XMLA users, then it implies that authentication will be performed using one of the underlying security mechanisms.

Regardless of what underlying security mechanism is used, the authentication is performed by the XMLA client software supplying the current Windows user credentials through a cryptographic exchange involving a single-sign-on (SSO) ticket (such as a Kerberos ticket). Then, the XMLA server should be able to use the username and the SSO ticket, which is submitted by the client to perform authentication and/or access control.

Minimal Connection Authentication

The cheapest form of authentication you can implement is HTTP authentication. This only requires setting up your web server; no addition is necessary in your ISO/JISO layers. Doing so nets you a coarse level of control: you can deny or grant access based on user credential; however, you won't be able to differentiate between users since no user credential has been passed thru to your ISO/JISO layer.

Session Security

Session security refers to securing the communication link between the XMLA client and the XMLA server. Since XMLA is a SOAP over HTTP web service, HTTPS can be used in place of HTTP to protect the datastream.

Consult the particular application server documentation on how to setup an HTTPS connection for your XMLA web service.

Appendix E – Client Programming

This appendix provides tips and pointers for using and/or developing client applications that connect to your OLE DB for OLAP or XML for Analysis provider. Due to the breadth of the topic, we will only be able to point out some common pitfalls. As typical for any programming problem, experimentation is crucial to understanding your scenario and/or problem.

ADO/ADOMD

ADO and/or ADOMD are arguably the most common approaches to OLE DB for OLAP because, being Microsoft technologies, they fit well together. They are sets of COM objects for accessing a data provider. ADO is largely geared for a relational/ SQL-aware data source while ADOMD is geared for a multi-dimensional/MDX-aware data source.

The OLE DB for OLAP standard anticipated a provider implementing both a relational and a multi-dimensional interface. For example, Microsoft SQL Server Analysis Services' ODBO provider can be used both as a multi-dimensional data provider (MDP) as well as a relational (or tabular) data provider (TDP).

SimbaProvider SDK is focused solely on multi-dimensional data access and includes only an MDX Engine for handling multi-dimensional data.

ADO may be used with a SimbaProvider-based provider but the functionality will be limited due to the absence of tabular data capabilities (metadata and command).

ADOMD is fully supported by any SimbaProvider-based provider. A good starting point for working with ADOMD is chapter 12 of the first edition of George Spofford's MDX Solutions.

Keep in mind that ADOMD was developed by Microsoft primarily for use with Microsoft Analysis Services 2000. Differences between any given provider and Microsoft Analysis Services 2000 will manifest themselves accordingly. That is, you may observe different behaviors, return values/results, or the occasional crashes. Do not assume that code and/or MDX that executes works with Analysis Services will behave identically for any other provider.

ADOMD.NET

For XML for Analysis, the most common development API at this time is ADOMD.NET.

As the name suggests, ADOMD.NET is a derivative of ADOMD targeted for .NET development. It is a .NET object model for working with an XML for Analysis provider. Being a .NET object model means that it is accessible from any .NET languages. A good starting point for working with ADOMD.NET is chapter 15 of the second edition of George Spofford's MDX Solutions.

ADOMD.NET was developed by Microsoft primarily for use with Microsoft Analysis Services 2000. The caveats above for ADOMD applies here too. Be on the lookout for differences between your provider and Microsoft Analysis Services, which will cause differences in behavior.