Magnitude Simba SDK

Build a C++ ODBC Driver for SQL-Capable Data Sources in 5 Days (Windows)

Version 10.2.2

October 2022

Copyright

This document was released in October 2022.

Copyright ©2014-2022 Magnitude Software, Inc., an insightsoftware company. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from Magnitude, Inc.

The information in this document is subject to change without notice. Magnitude, Inc. strives to keep this information accurate but does not warrant that this document is error-free.

Any Magnitude product described herein is licensed exclusively subject to the conditions set forth in your Magnitude license agreement.

Simba, the Simba logo, SimbaEngine, and Simba Technologies are registered trademarks of Simba Technologies Inc. in Canada, the United States and/or other countries. All other trademarks and/or servicemarks are the property of their respective owners.

All other company and product names mentioned herein are used for identification purposes only and may be trademarks or registered trademarks of their respective owners.

Information about the third-party products is contained in a third-party-licenses.txt file that is packaged with the software.

Contact Us

Magnitude Software, Inc.

www.magnitude.com

Table of Contents

About this Guide 5

Simba SDK Overview 8

- ODBC Standards 8
- The Simba SDK Solution 8
- About the UltraLight Sample Connector 9

Day One 14

- Install the Firebird data source 14
- Install the Simba SDK 14
- Build the Sample ODBC Connector 15
- Examine the Windows Registry 16
- View the Data Source Name (DSN) 21
- Connect to the Data Store 22
- Set Up a Custom ODBC Connector Project 23
- Update the Windows Registry 25
- Debug the Custom ODBC Connector 26
- Enable Logging 28

Day Two 30

- Finding the TODO messages 30
- Set the Configuration Branding 30
- Set Connector Properties 31
- Set Logging Details 32
- Check Connection Settings 33
- Customize the DriverPrompt Dialog 34
- Establish a Connection 35

Day Three 37

- Create and Return Metadata Sources 37

Day Four 43

- Prepare and Execute a Query 43

Day Five 46

- Rebrand Error Messages 46
- Rebrand the Custom ODBC Connector 47

Reference ... 49
- 32-bit vs 64-bit ODBC Data Source Administrator ... 49
- Bitness and the Windows Registry ... 50
- Data Retrieval ... 51
- Server Configuration ... 53
- Install the Evaluation License ... 53

Contact Us ... 55

Third-Party Trademarks ... 56

About this Guide

Purpose

This guide explains how to use the Magnitude Simba SDK to create a custom ODBC connector for a data store that is SQL-aware. It explains how to customize the UltraLight sample connector, which is included with the Simba SDK.

Using this sample connector is the quickest and easiest way to create a custom ODBC connector. At the end of five days, you will have a read-only connector that connects to your data store. This custom ODBC connector can be used as the foundation for a commercial DSI implementation.

Note:

An online version of this guide is located at http://www.simba.com/resources/sdk/documentation.

Advantages of Using the Simba SDK

The ODBC specification defines a rich interface that allows any ODBC-enabled application to connect to a data store. In order to implement a connector that supports this specification, developers have to understand all the complexities of error checking, session management, and data conversion, then design their code in a robust and efficient manner. Developers must also understand how to optimize data retrieval in order to get maximum performance when connecting to large and complex data stores.

The Simba SDK, developed by experts in the field, is a complete implementation of the ODBC specification. It exposes an easy-to-use SDK that allows you to create a robust and efficient connector for your data store.

Build a Custom ODBC Connector in Five Days

Over the course of five days, this guide explains how to accomplish the following tasks:

1. Set up the development environment and build the sample connector.
2. Use the sample connector as a template to create a custom ODBC connector.
3. Make a connection to the data store.
4. Retrieve metadata.
5. Work with columns.

6. Retrieve data.
7. Rename and rebrand the custom ODBC connector.

In the UltraLight connector, the areas of code that require modification are marked with “TODO” messages and a short explanation. Some of these changes customize the connector for your specific data store, while other changes rename the connector for your company or product.

Audience

The guide is intended for developers who want to use the Simba SDK to build a connector for a data store that is SQL-aware.

Document Conventions

Italics are used when referring to book and document titles.

Bold is used in procedures for graphical user interface elements that a user clicks and text that a user types.

`Monospace font` indicates commands, source code or contents of text files.

NOTE:

Indicates a short note appended to a paragraph.

IMPORTANT:

Indicates an important comment related to the preceding paragraph.

Knowledge Prerequisites

To use the Simba SDK to build a custom ODBC connector, the following knowledge is helpful:

- Familiarity with the C++ programming language.
- Ability to use the data store to which the connector you are developing will connect.
- An understanding of the role of ODBC technologies and driver managers in connecting to a data store.
- Exposure to SQL.

Variables Used in this Document

The following variables are used in this document:

Variable	Description
[INSTALL_DIR]	Installation directory for the SimbaEngine X SDK. **Default value on Windows platforms**: `C:\Simba Technologies\SimbaEngineSDK\10.2` **Default value on Linux, Unix, and macOS platforms**: `[UNTAR_DIR]/SimbaEngineSDK/10.2`
[UNTAR_DIR]	Directory where the SimbaEngine X SDK distributable was untarred.
[JDBC_VERSION]	The version of JDBC that your driver supports. You can use the SimbaEngine X SDK to build a driver for version 4.2 and 4.3, or a hybrid version. Possible values of *[JDBC_VERSION]* are `42` and `43`, and `Hybrid`.

Simba SDK Overview

Applications, such as Crystal Reports and Tableau, use connectors to connect to data stores from which they read and write data. Applications support the ODBC protocol to enable connection with any connector that also supports ODBC. A connector exposes the ODBC protocol to the application and another API, such as SQL or a custom API, to the data store.

Note:

This guide explains how to create an ODBC connector for a data store that is SQL-capable. To create an ODBC connector for a data store that is not SQL-capable, see Build a C++ ODBC Connector in 5 Days.

ODBC Standards

ODBC is one of the most established and widely-supported APIs for connecting to and working with databases. A main component of this technology is the ODBC connector, which connects an application to the database.

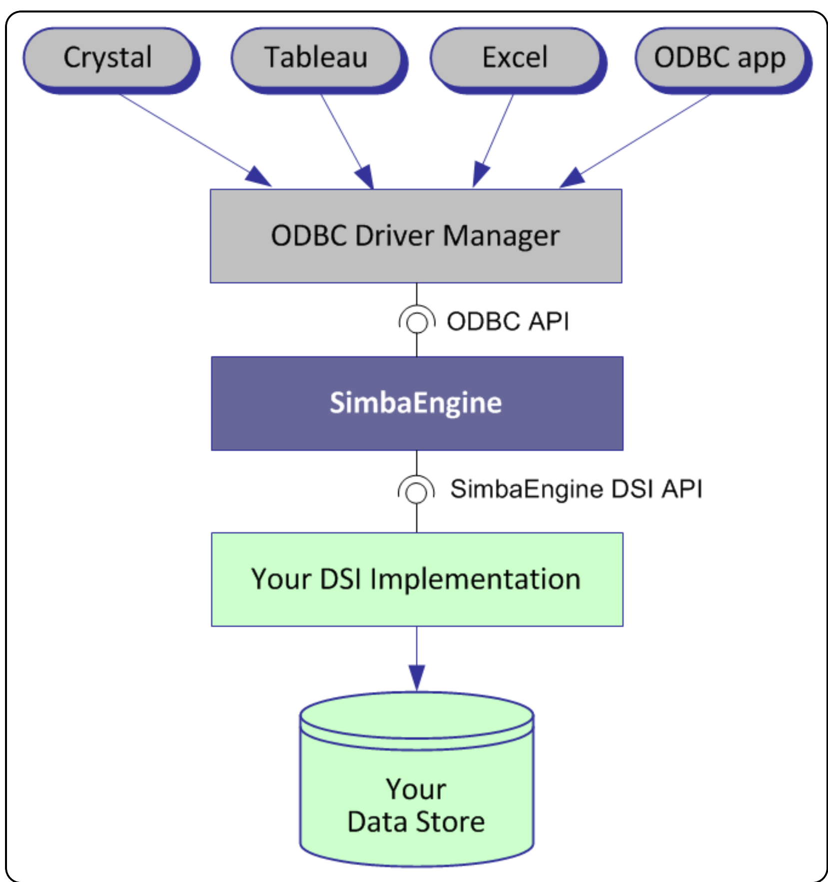
For a brief description of the ODBC standard, see http://www.simba.com/resources/data-access-standards-library#!odbc.

For complete information on the ODBC 3.80 specification, see the ODBC Programmer's Reference at http://msdn.microsoft.com/en-us/library/ms714177(v=vs.85).aspx.

The Simba SDK Solution

Connectors based on the Magnitude Simba SDK leverage its error checking, session management, data conversion, optimization, and other low-level implementation details. The Simba SDK uses ODBC to communicate with the driver manager and a simple API (called the Data Store Interface API or DSI API) to communicate with the data store. The DSI API defines the primitive operations needed to access a data store.

The figure below shows a typical ODBC stack:



SDK developers create an implementation of a DSI (also known as a DSI Implementation or DSII) that applications use to access the particular data store in the process of executing an SQL statement. In the final executable, the components from Simba SDK take responsibility for meeting the data access standards while the custom DSI implementation takes responsibility for accessing the data store and translating it to the DSI API.

ODBC applications, such as Tableau or Microsoft Excel, use this executable when connecting to the data store in the process of executing an SQL statement.

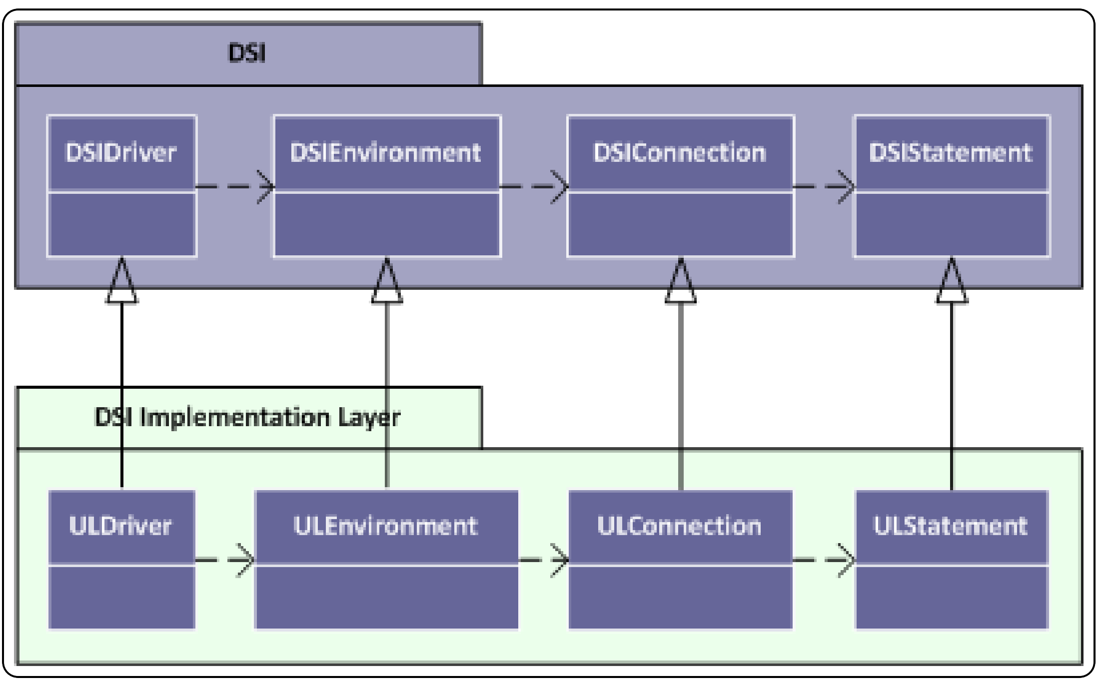
About the UltraLight Sample Connector

The Simba SDK includes a sample connector that you can use as a template to create a custom ODBC connector for data stores that are SQL-capable.The UltraLight connector is a sample DSI implementation of an ODBC connector, written in C++, which reads hard-coded data. The sample data is represented by a hard-coded table object, called the Person table. This table is always returned if an executed query contains SELECT. If the query does not contain SELECT, then a row count of 12 rows is returned.

Using the UltraLight sample connector to prototype a DSI implementation for a custom data store helps developers understand how the Simba SDK works. By removing the shortcuts and simplifications implemented in the UltraLight connector, you can use it as the foundation for a commercial DSI implementation and create a custom ODBC connector for a data store that is SQL-aware.

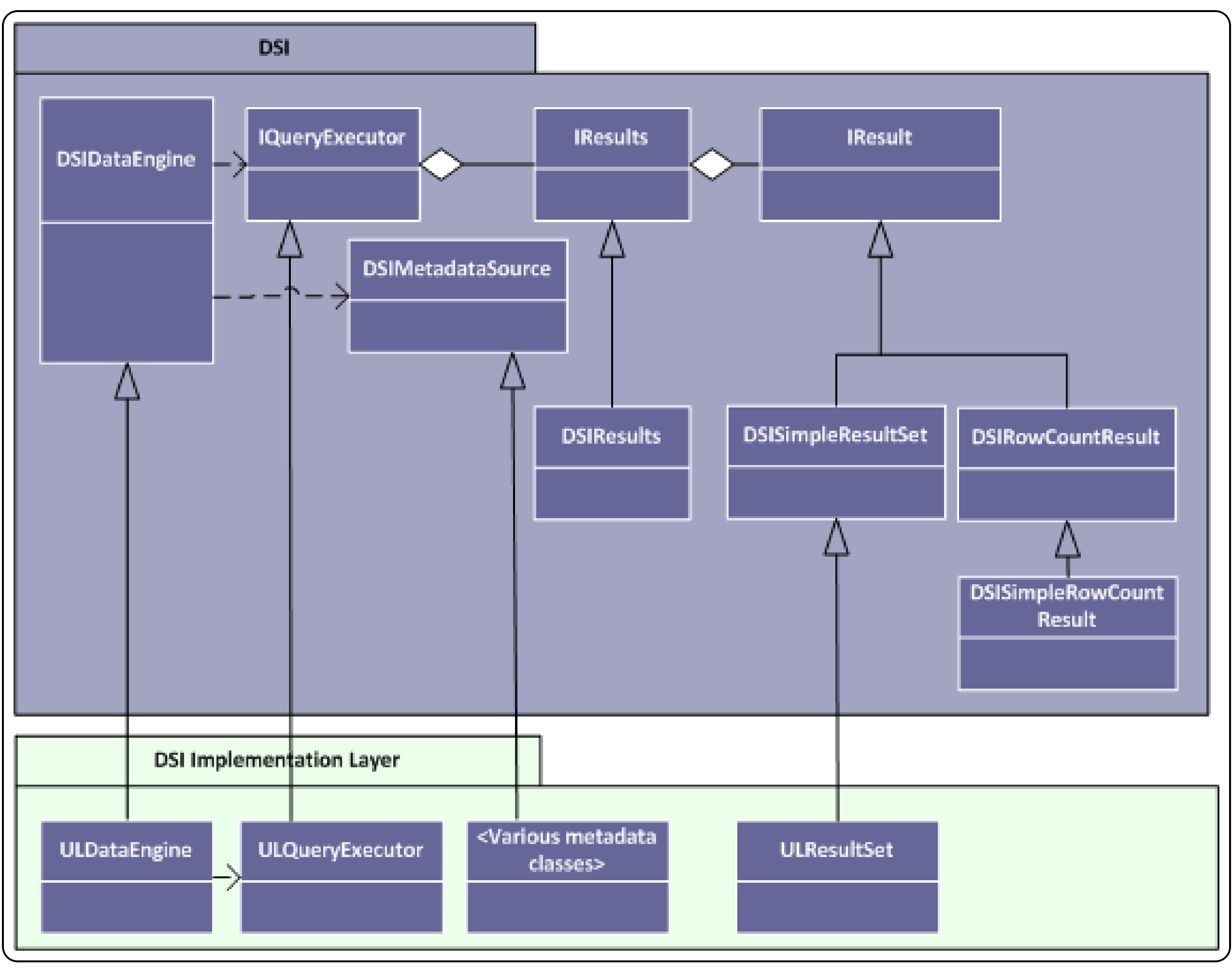
Implementation begins with the creation of a `DSIDriver` class which is responsible for constructing a `DSIEnvironment`. `DSIEnvironment` is used to construct a connection object (`DSIConnection` implementation) which is then used for constructing statements (`DSIStatement` implementations).

This concept is summarized in the figure below:

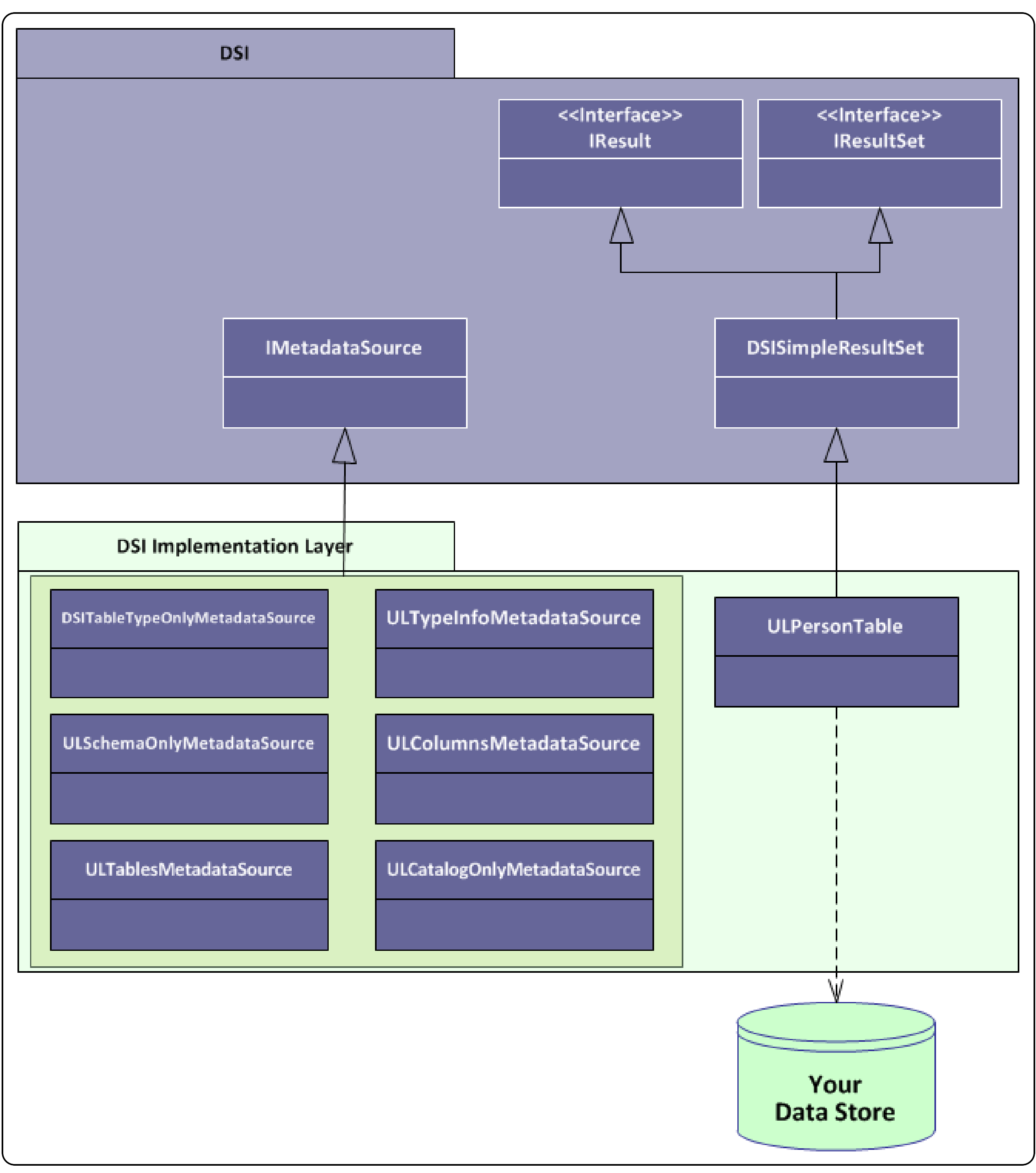


The `DSIStatement` implementation is responsible for creating a `DSIDataEngine` object, which then creates `IQueryExecutor` objects to execute queries and hold results (`IResults`), and `DSIMetadataSource` objects to return metadata information.

This concept is summarized in the figure below:



The final key part of the DSI implementation is to create the framework necessary to retrieve both data and metadata. A summary of this framework and the components implemented by the sample are shown in the figure below:



The `IResult` class is responsible for retrieving column data and maintaining a cursor across result rows.

To implement data retrieval, the custom `IResult` class interacts directly with the data store to retrieve the data and deliver it to the calling framework on demand. The

`IResult` class should take care of caching, buffering, paging, and all the other techniques that speed data access.

The various MetadataSource classes provide a way for the calling framework to obtain metadata information.

Day One

The Day One instructions explain how to install the Simba SDK, compile the sample ODBC connector, and review the configuration information created at compile time.

After the sample ODBC connector is successfully compiled, it is used to retrieve data from the data source that is included with the Simba SDK. The sample ODBC connector is then used to create the framework for a custom ODBC connector, which is renamed and used to retrieve sample data.

At the end of the day, you will have compiled, built and tested your custom ODBC connector.

Install the Firebird data source

The sample driver uses the open source database Firebird. You can download and install this software on your local machine for the purpose of this tutorial.

Firebird can be downloaded from this site: https://firebirdsql.org/en/server-packages/

Use the Windows executable installer package.

Instructions for installing Firebird are found here:
https://firebirdsql.org/file/documentation/reference_manuals/user_manuals/html/qsg3-installing.html

When installing Firebird:

1. Select the **Classic mode**installation.
2. On the Select Additional Tasks screen, accept all the default settings.
3. Ensure you make note of the Administrator user ID and password credentials you create during the installation, these will be required when testing the driver connection later.

Once you have completed installing the Firebird server, you may also wish to test that it is running using the instructions contained in the Firebird installation document.

Install the Simba SDK

The Simba SDK installation package includes:

- The Simba SDK implementation in all supported platforms.
- Sample connectors for most supported platforms.

- PDF versions of the documentation.
- An HTML version of the C++ and Java API.

To install the Simba SDK:

1. Uninstall any previous versions of the Simba SDK.
2. Make sure that Visual Studio is closed.
3. Double-click the Simba SDK executable that corresponds to your version of Visual Studio, and then follow the instructions provided in the installation wizard.

The installer sets the following environment variables, where `[INSTALL_DIR]` is the Simba SDK installation directory.:

Environment Variable	Value
SIMBAENGINE_DIR	*[INSTALL_DIR]*\SimbaEngineSDK\10.2\DataAccessComponents
SIMBAENGINE_THIRDPARTY_DIR	*[INSTALL_DIR]*\SimbaEngineSDK\10.2\DataAccessComponents\ThirdParty

Important:

The Simba SDK environment variables are defined only for the user who ran the installation. If the SDK is installed as a regular user, Visual Studio must also be run as a regular user and not an administrator.

Related Links

Install the Evaluation License on page 53

Build the Sample ODBC Connector

The UltraLight sample connector is included with the installation of the Simba SDK. It demonstrates one possible implementation of a read-only connector.

To build the UltraLight sample connector:

1. In Microsoft Visual Studio, click **File** > **Open** > **Project/Solution**.
2. In the Open Project dialog, navigate to the following folder:

`[INSTALL_DIR]\SimbaEngineSDK\10.2\Examples\Source\UltraLight\Source`

Where *[INSTALL_DIR]* is the installation directory.

3. Select the file `UltraLightDSII_vs2013.sln`, and then click **Open**.
4. Click **Build** > **Configuration Manager**.
5. Click the drop-down arrow next to the **Active Solution Configuration** field, then select **Debug_MTDLL**, and click **Close**.
6. Click the drop-down arrow next to the **Active Solution Platform** field:
   a. To build a 32-bit connector, select **Win32**.
   b. To build a 64-bit connector, select **x64**.
7. Click **Close**.
8. Click **Build** > **Build Solution**.

The build appears in the following folder:

`[INSTALL_DIR]\SimbaEngineSDK\10.2\Examples\Source\UltraLight\Bin\<BUILD>\<RELEASE|DEBUG><CONFIGURATION>`, where

- <*BUILD*> is a combination of your operating system, machine bitness, and compiler
- <*RELEASE|DEBUG*> is `release` or `debug`
- *CONFIGURATION*> is `mt` if you select MTDLL as the solution configuration, otherwise `md`

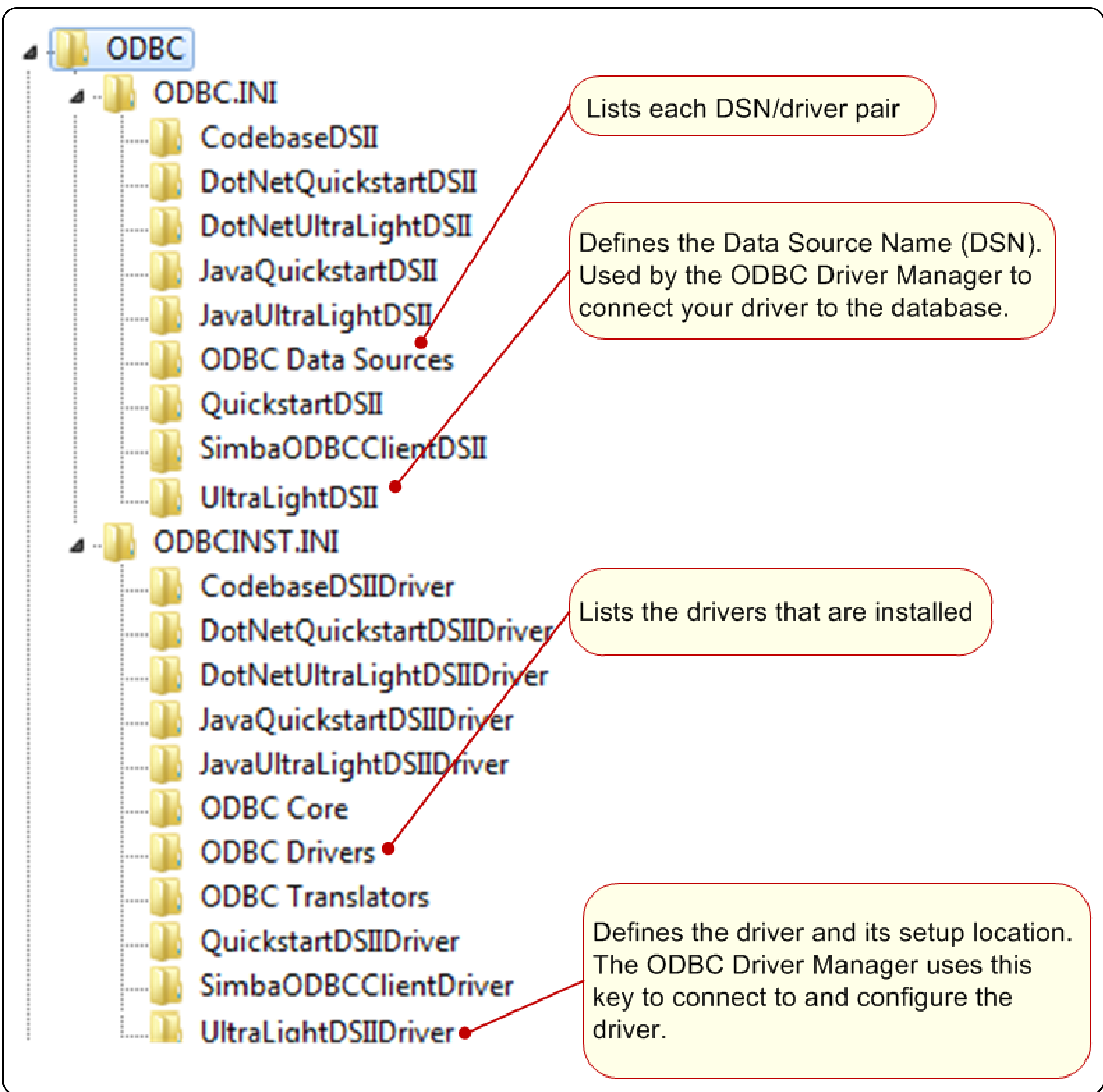
For example:
`C:\Simba Technologies\SimbaEngineSDK\10.2\Examples\Source\Ultralight\Bin\Windows_vs2013\debug32md\UltralightDSIIODBC32.dll`

Server Configuration on page 53

Examine the Windows Registry

The Simba SDK installer automatically adds or updates the following registry keys that define Data Source Names (DSNs) and connector locations:

- **ODBC Data Sources** - lists each DSN/connector pair
- **UltraLightDSII** - defines the Data Source Name (DSN). The ODBC Driver Manager uses this key to connect the connector to the database.
- **ODBC Drivers** - lists the connectors that are installed
- **UltraLightDSIIDriver** - defines the connector and its setup location. The ODBC Driver Manager uses this key to connect to and configure the connector.



Note:

The installer for your custom connector will create similar registry keys.

To view the registry keys for the UltraLight connector:

1. From a command line, run `regedit.exe`.
2. In the registry editor, navigate to one of the following root directories:
   - **HKEY_LOCAL_MACHINE\SOFTWARE\ODBC** for 64-bit connectors on 64-bit machines and 32-bit connectors on 32-bit machines.
   - Or, **HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432NODE\ODBC** for 32-bit connectors on 64-bit machines.
3. View the registry keys as explained in the rest of this section.

ODBC\ODBC.INI\UltraLightDSII key

This key defines the Data Source Name (DSN) for the UltraLight connector. It is located in the Windows Registry at:

- **HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\UltraLightDSII** for 64-bit connectors on 64-bit machines, and 32-bit connectors on 32-bit machines.
- **HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432NODE\ODBC\ODBC.INI\UltraLightDSIII** for 32-bit connectors on 64-bit machines.

This key has the following values:

Note:

Only the **Driver** subkey is required for your custom connector: other subkeys are specific to the UltraLight connector.

Subkey	Value	Description
Driver	UltraLightDSIIDriver	The name of the connector to use for connecting to the data store, as defined in the **ODBCINST.INI** key.
Description	Sample 32-bit Simba SDKUltraLight DSII Or, Sample 64-bit Simba SDK UltraLightDSII	A description of the DSN.

ODBC\ODBCINST.INI\ODBC Drivers key

This key contains one entry for every connector. The entry for the UltraLight connector is:

UltraLightDSIIDriver = Installed

ODBC\ODBCINST.INI\UltraLightDSIIDriver key

This key defines the UltraLight connector. It is located in the Windows Registry at:

- **HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\UltraLightDSIIDriver** for 64-bit connectors on 64-bit machines, and 32-bit connectors on 32-bit machines.
- **HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432NODE\ODBC\ODBCINST.INI\UltraLightDSIIDriver** for 32-bit connectors on 64-bit machines.

This key has the following values:

<table>
<tr><th>Subkey</th><th>Value</th><th>Description</th></tr>
<tr><td>Driver</td><td>C:\Simba Technologies\SimbaEngineSDK\10.2\Examples\Builds\Bin\<<i>BUILD</i>>\<<i>RELEASE|DEBUG</i>><<i>BITNESS</i>><<i>CONFIGURATION</i>>\UltralightDSIIODBC32.dll

Where:

• <<i>BUILD</i>> is a combination of your operating system, machine bitness, and compiler
• <<i>RELEASE|DEBUG</i>> is release or debug
• <i>CONFIGURATION</i>> is mt if you select MTDLL as the solution configuration, otherwise md

For example:
C:\Simba Technologies\SimbaEngineSDK\10.2\Examples\Builds\Bin\Windows_vs2013\release32md\UltralightDSIIODBC32.dll</td><td>The location of the connector DLL.

Note: this is the path to the pre-build DLL, not the DLL that you build in a previous step.</td></tr>
</table>

Subkey	Value	Description

Setup	`C:\Simba Technologies\SimbaEngineSDK\10.2\Examples\Builds\Bin\<BUILD>\<RELEASE|DEBUG><BITNESS><CONFIGURATION>\UltralightDSIIODBC32.dll`
	
	Where:
	• <*BUILD*> is a combination of your operating system, machine bitness, and compiler
	• <*RELEASE*|*DEBUG*> is `release` or `debug`
	• *CONFIGURATION*> is `mt` if you select MTDLL as the solution configuration, otherwise `md`
	
	For example:
	`C:\Simba Technologies\SimbaEngineSDK\10.2\Examples\Builds\Bin\Windows_vs2013\release32md\UltralightDSIIODBC32.dll`
		The location of the connector setup DLL.
		
		Note: this is the path to the pre-build DLL, not the DLL that you build in a previous step.

Description	Sample [32|64]-bit Simba SDK UltraLightDSII	A description of the connector.

View the Data Source Name (DSN)

The Windows ODBC Data Source Administrator can be used to view the DSNs for a connector, for example the UltraLight connector.

Ultralight does not have a DSN dialog. If you wish to see an example DSN dialog please refer to *Build a C++ ODBC Connector in 5 Days (Windows)*.

Connect to the Data Store

To connect to the data store and test the UltraLight connector, any ODBC application can be used. This section shows how to use the ODBCTest tool, which is included in the Microsoft Data Access (MDAC) 2.8 Software Development Kit (SDK): http://www.microsoft.com/downloads/details.aspx?FamilyID=5067faf8-0db4-429a-b502-de4329c8c850&displaylang=en

To connect to the data store using the UltraLight connector:

1. Navigate to the folder containing the ODBC Test application, by default: `C:\Program Files (x86)\Microsoft Data Access SDK 2.8\Tools`
2. Navigate to the folder that corresponds to your connector's architecture: **amd64**, **ia64** or **x86**.

   Example:

   If you built the 32-bit version of your connector on a 64-bit machine, select the **x86** version.
3. Click one:
   - **odbcte32.exe** to launch the ANSI version
   - Or, **odbct32w.exe** to launch the Unicode version.

   Important:

   It is important to run the correct version of the ODBC Test tool for ANSI or Unicode and 32-bit or 64-bit.
4. In the ODBC Test tool, click **Conn** > **Full Connect**.
   The Full Connect window opens.
5. In the Full Connect dialog, select **UltraLightDSII** from the list of data sources, and then click **OK**.
6. In the ODBC Test window, enter `SELECT * FROM ULResultSet`.
7. Click and to output a simple result set. The results are displayed in the window.

You have successfully used the UltraLight connector to connect to the sample data store and retrieve data.

Set Up a Custom ODBC Connector Project

Once the UltraLight has been built and tested, you can create a new project for your custom ODBC connector.

> **Important:**
>
> It is very important that you create your own project directory. You might be tempted to simply modify the sample project files, but we strongly recommend that you create your own project directory. If you simply modify the sample project files:
>
> - All your changes will be lost when you install a new version of the SDK.
> - You will lose your frame of reference for debugging.
>   There may be times, for debugging purposes, that you will need to see if the same error occurs using the sample connectors. If you have modified the sample connectors, this won't be possible.

To set up a custom project:

1. In Windows Explorer, copy the following directory and paste it to the same location:
   `[INSTALL_DIR]\SimbaEngineSDK\10.2\Examples\Source\UltraLight`

   where *[INSTALL_DIR]* is the Simba SDK installation directory. This will create a new directory called `UltraLight - Copy`.
2. Rename the directory to the name of your custom ODBC connector. This name will be referred to as [*PROJECT*] for the rest of these steps.
3. Rename the file `[PROJECT] > Source > UltraLightDSII_VS2013.vcxproj`. This is the project file for your custom ODBC connector.
4. Rename the `.sln` file. This is the solution file for your custom ODBC connector.
5. Using a text editor, open the project file `.vcxproj` and replace every instance of **UltraLightDSII** in the source code with the name of your custom ODBC connector.
6. Save and close the file `.vcxproj`.
7. Using a text editor, open the solution file `.sln` and replace every instance of **UltraLightDSII** in the source code with the name of your custom ODBC connector.
8. Change any references to the project file to `[PROJECT].vcxproj`.
9. Save and close the file.

Build the Custom ODBC Connector

To build the custom ODBC connector:

1. Launch Microsoft Visual Studio 2013.
2. Click **File**> **Open**>**Project/Solution**.
3. Navigate to `[INSTALL_DIR]\SimbaEngineSDK\10.2\Examples\Source\[PROJECT]\Source` and open the `[PROJECT].sln` file, where *[PROJECT]* is the renamed project.
4. Click **Build** > **Configuration Manager**, make sure that the active solution configuration is **Debug_MTDLL** and the active solution platform is **Win32**, then click **Close**.
5. Click **Build** > **Build Solution** to build the connector.
   This builds the Debug_MTDLL version of your custom ODBC connector and places the DLL in the following location:

   `[INSTALL_DIR]\SimbaEngineSDK\10.2\Examples\Source\[PROJECT]\Bin\<BUILD>\<RELEASE|DEBUG><CONFIGURATION>`, where

   - <*BUILD*> is a combination of your operating system, machine bitness, and compiler
   - <*RELEASE|DEBUG*> is `release` or `debug`
   - *CONFIGURATION*> is `mt` if you select MTDLL as the solution configuration, otherwise `md`
6. Make sure that the Output window is displayed. Select **Debug** > **Windows** > **Output**.

When the project for your custom ODBC connector is successfully built, the following “TODO” messages appear in the Output window along with the build information:

TODO #1: Construct connector singleton.
TODO #2: Set the connector properties.
TODO #3: Set the connector-wide logging details.
TODO #4: Set the connection-wide logging details.
TODO #5: Check Connection Settings.
TODO #6: Customize DriverPrompt Dialog.
TODO #7: Establish A Connection.
TODO #8: Create and return your Metadata Sources.
TODO #9: Prepare a Query
TODO #10: Implement a QueryExecutor.
TODO #11: Provide parameter information.
TODO #12: Implement Query Execution.

TODO #13: Implement your DSISimpleResultSet.
TODO #14: Register the ULMessages.xml file for handling by DSIMessageSource.
TODO #15: Set the vendor name, which will be prepended to error messages.

The instructions in the next four days explain how to modify the source code for each of the TODO messages.

Update the Windows Registry

The Data Source Name (DSN) and connector settings for the custom ODBC connector are configured in the Windows Registry. The custom connector configuration is similar to the UltraLight connector configuration, described in Examine the Windows Registry on page 16.

The Simba SDKncludes `.reg` files that you can modify for your custom ODBC connector, then use to create the connector's registry keys.

To update the Windows Registry:

1. In Microsoft Visual Studio or another text editor, navigate to the following directory:
   `[INSTALL_DIR]\SimbaEngineSDK\10.2\Examples\Source\[PROJECT]\Source`

   where [*INSTALL_DIR*] is the installation directory and *[PROJECT]* is the name of your custom connector project.
2. Open one of the following files:
   - For 32-bit Windows, open `SetupMyUltraLightDSII-32on32.reg`.
   - Or, for a 32-bit ODBC connector on 64-bit Windows, open `SetupMyUltraLightDSII-32on64.reg`.
   - Or, for a 64-bit ODBC connector on 64-bit Windows, open `SetupMyUltraLightDSII-64on64.reg`.
3. In the file, replace `[INSTALL_DIRECTORY]` with the Simba SDK installation directory. Use double backslashes in the path.

   Example:

   If the Simba SDK installation directory is `C:\Simba Technologies`, replace all instances of `[INSTALL_DIRECTORY]` with `C:\\Simba Technologies`.
4. Change all instances of `MyUltraLight` to the name of your custom ODBC connector. The following changes will be made:
   - Under **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\ODBC Data Sources]**, the

key `"MyUltraLightDSII"="MyUltraLightDSIIDriver"`

- "Driver"="MyUltraLightDSIIDriver"
- **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\MyUltraLightDSII]**
- Under **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers]**, the key `"MyUltraLightDSIIDriver"="Installed"`
- **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\MyUltraLightDSIIDriver]**

5. Update **"Setup"** with the path to the custom connector DLL.
6. Update **"Driver"** with the path to the custom connector DLL.
7. Replace every other instance of `UltraLight` in the file with the name of your custom ODBC connector.
8. Click **Save** and close the file.

Note:

Some of the registry settings in this `.reg` file are used for Client - Server configuration. If you are not developing a connector for Client - Server deployment, you can choose to either remove these keys or keep them in.

To import the configuration settings into the Windows Registry:

1. Open `regedit.exe`, click **File** > **Import**, navigate to the registry file that you modified, and then click **Open**.
   A message indicating that the keys and values have been successfully added to the registry is displayed.
2. To verify the Data Source Name for the new project, see View the Data Source Name (DSN) on page 21.

Debug the Custom ODBC Connector

You can use the Visual Studio debugger to step through your custom ODBC connector code in order to gain a better understanding of its functionality. This section explains how to use the ODBCTest application to connect to your custom ODBC connector, then use Visual Studio debugger to step through the connector code.

To open the ODBC Test application:

1. Navigate to the folder containing the ODBC Test application, by default: `C:\Program Files (x86)\Microsoft Data Access SDK 2.8\Tools`
2. Navigate to the folder that corresponds to your machine's architecture: **amd64**, **ia64** or **x86**.
3. Click one:
   - **odbcte32.exe** to launch the ANSI version
   - Or, **odbct32w.exe** to launch the Unicode version.

   Important:

   It is important to run the correct version of the ODBC Test tool for ANSI or Unicode and 32-bit or 64-bit.

To attach the Visual Studio debugger to the ODBC Test process:

1. In Microsoft Visual Studio, click **Debug**>**Attach to Process**.
2. In the Attach to Process window, select the ODBC test process launched in the previous step, and then click **Attach**.

   The process name will be either `odbc32.exe` or `odbct32w.exe`.
3. Add a breakpoint in `Main_Windows.cpp`, on the function `DSIDriverFactory()`.

   This function runs as soon as the Driver Manager loads the ODBC connector.
4. In the ODBC Test tool, select **Conn**> **Full Connect**. The Full Connect window opens.
5. Select the data source from the list of data sources and then click **OK**.
6. **Note:**

   If you do not see the custom data source in the list, make sure that you are running the version of the ODBC Test tool that corresponds to the version of the data source. For example, use a 32-bit version of the ODBC Test tool to connect to a 32-bit data source.
7. The Visual Studio debugger hits the breakpoint created at `DSIDriverFactory()`.
8. To continue running the program, select **Debug**>**Continue**.The focus returns to the ODBC Test window.

9. In the ODBC Test window, enter `SELECT * FROM ULResultSet`.
10. Click [icon] and [icon] to output a simple result set.

This step verifies that your custom ODBC connector, based on the UltraLight project, is correctly installed and configured, and that the development system is properly set up.

Summary of Day One

You have successfully completed the following tasks:

- Built and tested the UltraLight sample connector.
  This verifies that your installation and development environment are properly configured.
- Created, built, and tested a custom connector project by copying the UltraLight connector.
  You can use this project as a framework to create your custom ODBC connector.

Related Links

ODBC Troubleshooting: How to Enable Driver-manager Tracing

Testing ODBC Connectors On Windows in Developing Connectors for SQL-capable Data Stores

Enable Logging

You can turn on logging for your custom ODBC connector. By setting the log level to `Trace`, you can gain a better understanding of how your custom ODBC connector works.

To enable logging in your custom ODBC connector:

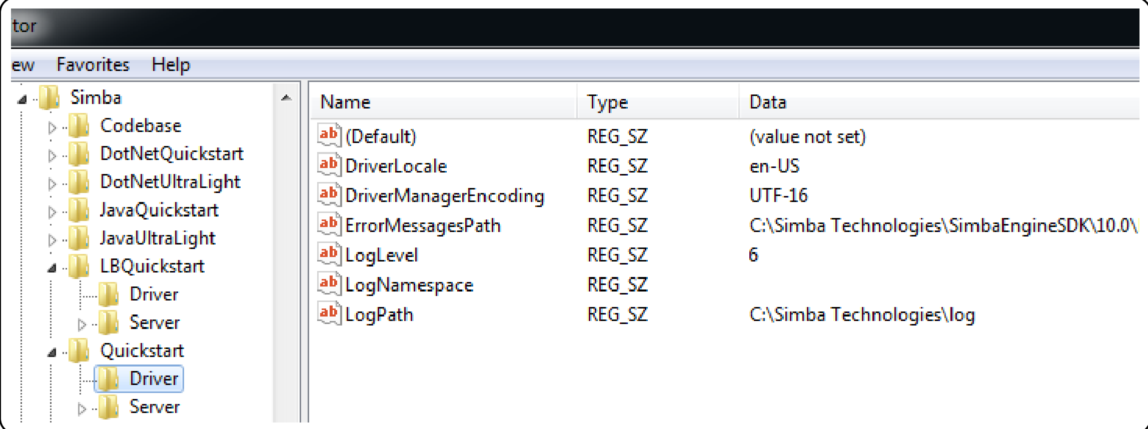
1. Use `regedit.exe` to open the registry editor.
2. Navigate to one of the following keys:
   - **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Simba\UltraLight\Driver** for 32-bit connectors on 64-bit machines
   - **Or, HKEY_LOCAL_MACHINE\SOFTWARE\Simba\UltraLight\Driver** for 32-bit connectors on 32-bit machines, or 64-bit connectors on 64-bit machines

Note:

Your custom ODBC connector has not been completely rebranded yet, so some information is still read from the **Simba\Ultralight** key.

3. In the **LogLevel** key, set the value to `6` for trace level, or another level if you prefer.
4. In the **LogPath** key, configure the directory to use for log files.

   For example, if your custom ODBC connector is based on the UltraLightconnector and not rebranded:



5. If you are using an ODBC test application with your custom ODBC connector, restart the application to reload the new connector settings.

The log files are created the next time the connector is used.

Log File Format

The log files have the following format, where `[Message]` is optional:

```
[Date] [Log Level] [Thread ID] [Class] [Message]
```

Example:

```
Jun 15 14:05:12.017 INFO 9864
ConnectionSettings::LoadSettings: ConnString setting: "DSN" =
"MyQuickstartDSII"
```

Implementing Logging in Developing Connectors for SQL-capable Data Stores

http://www.simba.com/resources/sdk/knowledge-base/enable-logging-in-odbc/

http://www.simba.com/resources/sdk/knowledge-base/simbaengine-logging/

Day Two

Day Two instructions explain how to customize your ODBC connector, enable logging, and establish a connection to your data store.

Finding the TODO messages

When the custom project is built, TODO messages display in the Output window.

To rebuild the whole solution, select **Build** > **Rebuild Solution**. If the Output window is not open, select **Debug** > **Windows** > **Output**.

Double click a TODO message to jump to the relevant section of code.

Set the Configuration Branding

The `DSIDriverFactory()` implementation in `Main_Windows.cpp` is the main entry point that is called from Simba's ODBC layer to create an instance of the DSI implementation. This method is called as soon as the Driver Manager calls `LoadLibrary()` on the ODBC connector DLL.

To construct the connector singleton:

1. In your custom ODBC connector project, double-click the **TODO #1 Construct connector singleton** message to jump to the relevant section of code. The `Main_Windows.cpp` file opens.
2. Look at the `DSIDriverFactory()` implementation.
3. Rebrand the connector name and the company name. This change affects the location in the Windows Registry where the connector values are looked up:
   a. Find the line `SimbaSettingReader::SetConfigurationBranding (DRIVER_WINDOWS_BRANDING);`
   b. Right click `DRIVER_WINDOWS_BRANDING` and select **Go to Definition**. The file `UltraLight.h` opens.
   Note: The file name might have a different case, for example `UltraLight.h` or `Ultralight.h`.
   c. In the line `#define DRIVER_WINDOWS_BRANDING "Simba\\UltraLight"`, change `Simba` to the company name and `UltraLight` to the custom connector name.

   Example:

   If `DRIVER_WINDOWS_BRANDING` is set to `"Simba\\UltraLight"`, then the base path for values in the Windows Registry is

- **HKLM\SOFTWARE\Simba\UltraLight** for 32-bit connectors on 34-bit Windows, or 64-bit connectors on 64-bit Windows
- Or, **HKLM\SOFTWARE\SOFTWARE\Wow6432Node\Simba\UltraLight** for 32-bit connectors on 64-bit Windows

If the DSII is compiled as a connector, it will use **\Driver** as a suffix. If the DSII is compiled as a server, it will use **\Server** as a suffix.

Therefore, a 64-bit connector would use the full path of **HKLM\SOFTWARE\Simba\UltraLight\Driver** to look up the registry keys such as **ErrorMessagesPath**.

4. Optionally, add processing at this point for building a commercial connector.
5. Click **Save**.

Set Connector Properties

To set connector properties:

1. Double click the **TODO #2 Set the connector properties** message to jump to the relevant section of code. The `ULDriver.cpp` file opens.
2. Open the file `ULDriver.cpp` file and navigate to the line **TODO #2 Set the connector properties**.
3. Go to the method `SetDriverPropertyValues()`, where the general properties for the connector are set. Change the properties described below:

Property	Description
DSI_DRIVER_DRIVER_NAME	Set this property to the name of the connector (the same name used to replace UltraLightDSII in Day One). This is the connector name that is shown to the application.

Property	Description
DSI_DRIVER_STRING_ DATA_ENCODING	Optional. The encoding of char data from the perspective of the data store. Depending on the character sets, this property may need to be changed.
DSI_DRIVER_WIDE_STRING_ DATA_ENCODING	Optional. The encoding of wide character data from the perspective of the data store. Depending on the character sets, this property may need to be changed.

Set Logging Details

This section explains how to set the connector-wide and connection-wide logging.

To set logging details:

1. Double click the **TODO #3 Set the connector-wide logging details** message to jump to the relevant section of code.
2. Change the connector log's file name.
3. Double click the **TODO #4 Set the connection-wide logging details** message to jump to the relevant section of code.
4. By default, the connections use the same log file as the connector. If connections and connectors require separate log files, change the code to create a DSILog with a unique log file name.
5. Click **Save All**.

Note:

Note: By default, the UltraLight connector maintains one log file for the entire connector. If you require more fine grained logging, consider implementing one log file for all connector-based calls and one log file for each connection created.

For more information about how to enable logging, see Developing Connectors for SQL-capable Data Stores.

Check Connection Settings

When the Simba ODBC layer is given a connection string from an ODBC-enabled application, the Simba ODBC layer parses the connection string into key-value pairs. The entries in the connection string and the DSN are then sent to the `ULConnection::UpdateConnectionSettings()` function for validation.

If entries of the connection string overlap entries from the DSN, then the connection string will override parameters from the DSN.To pass additional parameters to your DSII, simply add new parameters to the connection string, or add new entries to the DSN entry. These values will automatically be picked up by the SDK and passed through for use by your DSII.

`UpdateConnectionSettings()` receives all the incoming connection settings that are specified in the DSN that was used to establish the connection. The role of this function is to ensure that all of the required, and any optional, settings are present. Note that actual data validation of the settings should be done in the `Connect()` function.

Example:

The connection string “DSN=UltraLight;UID=user;” will be broken down into key value pairs and passed in via the `DSIConnSettingRequestMap` parameter. In this case that map would contain two entries: {DSN, UltraLight} and {UID, user}. If a DSN was specified, then the DSN value is removed from the map and any entries that are stored in the preconfigured DSN are inserted into the map. Once the map has been created with all the key-value pairs from the connection string and DSN, this map is passed down to the DSII.

To check the connection settings for the custom connector:

1. Double click the **TODO #5 Check Connection Settings** message to jump to the relevant section of code.
2. Modify the `UpdateConnectionSettings()` function to validate that the settings (key-value pairs) in the `DSIConnSettingRequestMap` are sufficient to create a connection. Any settings that are not present should be added to the `DSIConnSettingResponseMap` parameter.

   We recommend using the `VerifyRequiredSetting()` or `VerifyOptionalSetting()` functions to perform this verification. These functions also add missing settings to `DSIConnSettingResponseMap`.

Note:

The connection settings listed in `UpdateConnectionSettings()` are specific to the UltraLight connector. A custom connector will require different settings.

Example - UltraLight Connector

The UltraLight connector verifies that the settings contained in `in_connectionSettings` are sufficient to create a connection, by using the following code:

```
VerifyRequiredSetting(UL_UID_KEY, in_connectionSettings,
out_connectionSettings);
VerifyRequiredSetting(UL_PWD_KEY, in_connectionSettings,
out_connectionSettings);
VerifyOptionalSetting(UL_LNG_KEY, in_connectionSettings,
out_connectionSettings);
```

The UltraLight connector requires a user ID and password, and can optionally take in a language (not currently used).

3. If any required values are missing, the connector will either fail to connect, or will call `PromptDialog()`, depending on the connection settings. If all required values exist, then `Connect()` will be called.
4. If any of the values received are invalid, then the code should throw an `ErrorException` seeded with `DIAG_INVALID_AUTH_SPEC`.

Manually verifying the connection settings

Settings can also be verified manually. If the entries within `in_connectionSettings` are not sufficient to create a connection, then the connector can ask for additional information from the ODBC-enabled application by manually specifying the additional, required settings in `out_connectionSettings`. If there are no further entries required, simply leave `out_connectionSettings` empty.

For more information on ODBC connections, see the Knowledge Base article *DSII Connection Process for ODBC* at http://www.simba.com/resources/sdk/knowledge-base/dsii-connection-process-for-odbc.

Customize the DriverPrompt Dialog

The Simba SDK may call `ULConnection::PromptDialog()` to display a dialog box that prompts the user for information about the connection. In general, `PromptDialog()` is called if there are any required settings in `DSIConnSettingResponseMap`. However, the application may request that

`PromptDialog()` is not called when required settings exist in `DSIConnSettingResponseMap`, or the application may request that `PromptDialog()` is called even if there are no required settings in `DSIConnSettingResponseMap`.

Double-click the **TODO #6 Customize DriverPrompt Dialog** message to jump to the relevant section of code.

`ULConnection::PromptDialog()` displays a configuration dialog box, which is displayed by the Windows ODBC Data Source Administrator when configuring the connector.

The method takes the following parameters:

- `in_connResponseMap`: a connection response map that contains missing values. Missing values are required in order to establish the connection, but were not supplied by the user. The connector uses this map to notify the user about missing information. The sample ODBC connector does not use this parameter.
- `io_connectionSettings`: a connection settings map containing key-value pairs that are used to populate the dialog box. This map is updated with the user's input to the dialog box.
- `in_parentWindow`: the handle to the parent window to which the dialog belongs.
- `in_promptType`: indicates whether only required settings are to be available in, or both optional and required settings. In the UltraLight connector, the language is an optional field.

You can modify this method to handle different parameters, as required by your custom ODBC connector.

Establish a Connection

The Simba SDK calls `UpdateConnectionSettings()` before calling `ULConnection::Connect()`. Once `ULConnection::UpdateConnectionSettings()` returns `out_connectionSettings` without any required settings–if there are only optional settings, a connection can still occur–then the Simba ODBC layer calls `ULConnection::Connect()`, passing in all the connection settings received from the application.

During `Connect()`, the connector should have all the settings necessary to make a connection as verified by `UpdateConnectionSettings()`. You can use the utility functions `GetRequiredSetting()` and `GetOptionalSetting()` to request the

required and optional settings for your connection, and attempt to make an actual connection.

To establish a connection:

1. Double click the **TODO #7 Establish A Connection** message to jump to the relevant section of code.
2. Look at the code that authenticates the user against your data store using the information provided within the `in_connectionSettings` parameter. Use `GetRequiredSetting()` and `GetOptionalSetting()` to access the settings in the map.
3. Add validation to your custom ODBC connector. If authentication fails, throw an ErrorException seeded with `DIAG_INVALID_AUTH_SPEC`. Note that the sample ODBC connector does not perform validation.

The user is now authenticated against your data store.

Summary of Day Two

You have successfully authenticated the user against your data store and established a connection.

Day Three

The Day Three instructions explain how to return the data used to pass catalog information back to the ODBC-enabled application.

Create and Return Metadata Sources

Your custom ODBC connector uses metadata sources, provided by the Simba SDK, to handle SQL catalog functions.

Overview of SQL Catalog Functions

ODBC applications need to understand the structure of a data store in order to execute SQL queries against it. This information is provided using catalog functions. For example, an application might request a result set containing information about all the tables in the data store, or all the columns in a particular table. Each catalog function returns data as a result set.

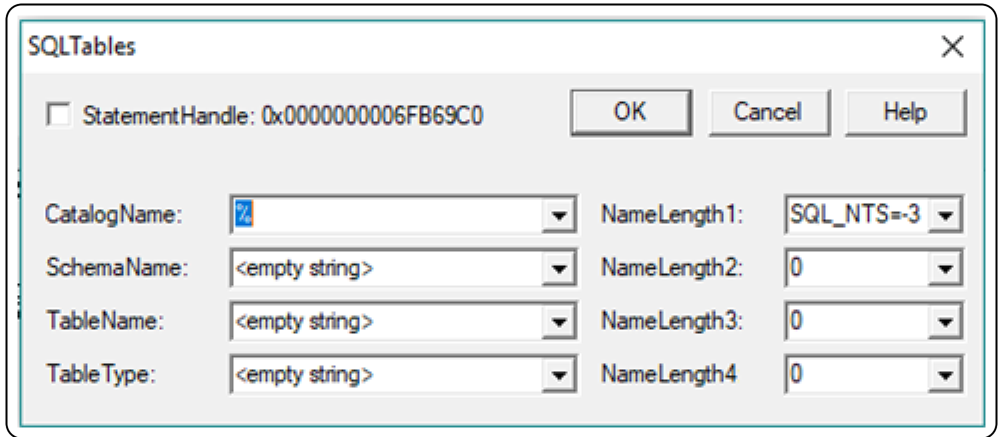
Most ODBC-enabled applications require a connector to implement the following catalog functions.You may wish to implement additional catalog functions in your custom connector.

Catalog Function	Description
SQLGetTypeInfo	Returns information about data types supported by the data source.
SQLTables (CATALOG_ONLY)	If CatalogName is SQL_ALL_CATALOGS and SchemaName and TableName are empty strings, the result set contains a list of valid catalogs for the data source. (All columns except the TABLE_CAT column contain NULLs.)
SQLTables (SCHEMA_ONLY)	If SchemaName is SQL_ALL_SCHEMAS and CatalogName and TableName are empty strings, the result set contains a list of valid schemas for the data source. (All columns except the TABLE_SCHEM column contain NULLs.)

Catalog Function	Description
SQLTables (TABLE_TYPE_ONLY)	If TableType is SQL_ALL_TABLE_TYPES and CatalogName, SchemaName, and TableName are empty strings, the result set contains a list of valid table types for the data source. (All columns except the TABLE_TYPE column contain NULLs.)
SQLTables	Returns the list of table, catalog, or schema names, and table types, stored in a specific data source.
SQLColumns	Returns a list of columns in one or more tables.

Example: Using Catalog Functions with the UltraLightconnector

1. In the ODBC Test application, connect to the UltraLight connector.
2. To send the SQLTables (CATALOG_ONLY) catalog function, select **Catalog** > **SQLTables**.
3. Enter SQL_ALL_CATALOGS for the **CatalogName**, then select the correct value for **NameLength1**. For example:



4. Click **OK**.
5. Select to retrieve the results.

The following list of valid catalogs for the UltraLight data source are returned:
`"TABLE_QUALIFIER", "TABLE_OWNER", "TABLE_NAME", "TABLE_TYPE", "REMARKS"`

For more information on SQL catalog functions, see https://msdn.microsoft.com/en-us/library/ms713520(v=vs.85).aspx.

Implementing Metadata Sources to Handle Catalog Functions

SQL catalog functions are represented in the DSI by metadata sources: there is one metadata source for each of the catalog functions.

`ULDataEngine::MakeNewMetadataTable()` is responsible for creating the metadata sources. Metadata sources are used return the catalog metadata about your data store to the ODBC application for the ODBC catalog functions.

Double click the **TODO #8 Create and return your Metadata Sources** message to jump to the relevant section of code.

There is one metadata source for each of the catalog functions. For example, when the application calls `SQLColumns()`, a `DSI_COLUMNS_METADATA` source is created to return the list of columns in one or more tables in the data store.

Each ODBC catalog function is mapped to a unique `DSIMetadataTableId`, which is then mapped to an underlying `MetadataSource` that the connector implements and returns. Each `MetadataSource` instance is responsible for the following:

1. Creating a data structure that holds the data relevant for the custom data store: `Constructor`
2. Navigating the structure on a row-by-row basis: `Move()`
3. Retrieving data: `getMetadata()` (See Data Retrieval on page 51 for a brief overview of data retrieval). Each column in the metadata source will be represented by a `DSIOutputMetadataColumnTag`, which is passed into `GetMetadata()`.

Required Metadata Sources

All custom ODBC connectors must implement the following metadata sources, as they are required by ODBC applications:

Metadata Source	Description
DSI_TABLES_METADATA	List of all tables defined in the data source.
DSI_CATALOGONLY_ METADATA	List of all catalogs defined in the data source, if catalogs are supported.
DSI_SCHEMAONLY_ METADATA	List of all schemas defined in the data source. This source is constructed via the ULMetadataHelper and SQL Engine.

Metadata Source	Description
DSI_TABLETYPEONLY_METADATA	List of all table types (TABLE,VIEW,SYSTEM) defined within the data source.
DSI_COLUMNS_METADATA	List of all columns defined across all tables in the data source.
DSI_TYPE_INFO_METADATA	List of the supported types by the data source. This means the actual types that can be stored in the data source, not necessarily the types that can be returned by the connector. For instance, a conversion may result in a type being returned that is not stored in the data source.

Most catalog types are created using the metadata helper.

Handling DSI_TYPE_INFO_METADATA

The underlying ODBC catalog function `SQLGetTypeInfo` is handled as follows:

1. When called with DSI_TYPE_INFO_METADATA, `ULDataEngine::MakeNewMetadataTable()` will return an instance of `ULTypeInfoMetadataSource()`.
2. The UltraLight sample connector exposes support for all data types, but due to its underlying file format, it is constrained to support only the following types:

- SQL_BIGINT
- SQL_CHAR
- SQL_DOUBLE
- SQL_LONGVARCHAR
- SQL_REAL
- SQL_TYPE_DATE
- SQL_VARBINARY
- SQL_WVARCHAR
- SQL_NUMERIC
- SQL_TINYINT
- SQL_BINARY
- SQL_DECIMAL
- SQL_INTEGER
- SQL_LONGWVARCHAR
- SQL_SMALLINT
- SQL_TYPE_TIME
- SQL_VARCHAR
- SQL_BIT
- SQL_FLOAT
- SQL_LONGVARBINARY
- SQL_TYPE_TIMESTAMP

- SQL_WCHAR

3. For your connector, you may need to change the types returned and the parameters for the types in `ULTypeInfoMetadataSource::InitializeData()`. Populate the `m_dataTypes` vector in this method, which defines the collection types that are supported along with their parameters.

Handling the Other MetadataSources

The other ODBC catalog functions, including SQLTables (CATALOG_ONLY), SQLTables (TABLE_TYPE_ONLY), SQLTables (SCHEMA_ONLY), SQLTables and SQLColumns, are handled automatically by the metadata helper class.

When these functions are called with the corresponding metatable ID's, `ULDataEngine::MakeNewMetadataTable()` returns a new instance of one of the following `DSIMetadataSource`-derived classes:

- `ULCatalogOnlyMetadataSource`: returns a list of all catalogs. The sample implementation returns one row of information with one column containing the name of a catalog. While the catalog does not actually exist in the sample connector, this demonstrates how to return a catalog name.
- `DSITableTypeOnlyMetadataSource`: In the sample implementation, this returns metadata about all tables of a particular type (TABLE, SYSTEM TABLE, and VIEW) in the datasource. This class provides two constructors, which allow for returning the default set of table types (listed above) or for specifying your own set of table types.
- `ULSchemaOnlyMetadataSource`: returns a list of all schemas. The sample implementation returns one row of information with one column containing the name of a fake schema. This demonstrates how to return a schema name.
- `ULTablesMetadataSource`: returns metadata about all of the tables in the data source. The sample hard codes and returns information for the hard coded person table to demonstrate how to return table metadata.
- `ULColumnsMetadataSource`: returns metadata for the columns in the data source. The sample hard codes and returns information for the three columns in the person table consisting of the name column, an integer column, and a numeric column.

When called with any other `DSIMetadataTableId`, which doesn't correspond to these tables, `ULDataEngine::MakeNewMetadataTable()` returns a new instance of `DSIEmptyMetadataSource` to indicate that no metadata is available for the specified table ID.

You can now retrieve type metadata from your data store.

For more information on the other metadata source types, see the `DSIMetadataTableId.h` header file.

Fetching Metadata for Catalog Functions in Developing Connectors for SQL-capable Data Stores

Summary of Day Three

Your custom ODBC connector can now return type metadata. You can use a ODBC-enabled application to connect to your connector and retrieve type metadata from within your data store

Day Four

Day Four instructions explain how to enable data retrieval from within the connector.

Prepare and Execute a Query

This section explains how to prepare a query, provide parameter information, implement a query executor, and create a result set.

Prepare a Query

Click on **TODO #9: Prepare a Query** to go to the relevant section of code.

The `ULDataEngine::Prepare()` method takes in a query and passes it to the underlying SQL-enabled data source for preparation. Once the query is prepared, the method returns a `ULQueryExecutor` that is used by the engine to return results.

For demonstration purposes, the UltraLight implementation of `ULDataEngine::Prepare()` performs a very simple preparation by searching for the substrings “SELECT” and “?” in the query. If “SELECT” is found, then it is assumed that the caller wants to search for rows of data and a result set is returned. If “SELECT” is not found, then it is assumed that the caller wants to retrieve the number of rows and so a row count is therefore returned. If “?” is present, then the statement is assumed to be parameterized and therefore `ULDataEngine::PopulateParameters()` will populate parameters as described below.

In your implementation, replace this section with functionality for your custom ODBC connector, or pass the query to the data source for preparation.

Implement a Query Executor

Click on **TODO #10: Implement an IQueryExecutor** to jump to the relevant section of code.

The `ULQueryExecutor` object returned by the `ULDataEngine::Prepare()` method is an implementation of `IQueryExecutor`, which executes a query. After preparing a query, the application can execute it multiple times. If the application executes a query multiple times, a single `IQueryExecutor` is created and used for each execution.

The UltraLight implementation of `ULQueryExecutor` checks if the query passed in contains a SELECT statement. If `in_isSelect` is set, the constructor creates and adds a simple result set consisting of people’s names to `m_results`. Otherwise, it creates and adds a simple row count.

In your custom ODBC connector, you can move the retrieval and storage of the result set out of this method and into `ULQueryExecutor::GetResults()`.

Note that `GetResults()` is called before query execution to retrieve and inspect the result set's metadata. This is because ODBC allows applications to retrieve column metadata from a query before execution, although the metadata does not need to be accurate until after execution.

Modify the implementation to query the data source and store the results.

Provide Parameter Information

Click on **TODO #11: Provide Parameter Information** to jump to the relevant section of code.

The `ULQueryExecutor::PopulateParameters()` method is where parameter information is specified when the application calls `SQLPrepare`. The default implementation shows how to register input, input/output, and output only parameters. Modify this method to register parameters that are appropriate for your custom ODBC connector queries.

Note that this method will only be called if `ULQueryExecutor::GetNumParams()` indicates that there is at least one parameter in the query and if the hosting application doesn't set SQL_ATTR_ENABLE_AUTO_IPD to `false`.

Implement Query Execution

Click on **TODO #12: Implement Query Execution** to jump to the relevant section of code.

The next step is to handle statement execution in `ULQueryExecutor::Execute()`. The UltraLight implementation simply resets the results obtained in the constructor in preparation for the application to retrieve them. If the executor is handling a parameterized statement, then additional logic iterates through the input and copies it to the output for consumption by the calling application.

In your implementation, the `Execute()` method should begin by serializing parameters (stored in `in_inputParamSetter`) into a form that the data source can consume. Once this has been done, the data source should be instructed to execute the statement. After the statement is executed, the results should be placed into the `in_outputParamSetIter parameter`.

After this method exits, the calling framework will then invoke `ULQueryExecutor::GetResults()` to obtain the result set.

Implement the Result Set

Click on **TODO #13: Implement your DSISimpleResultSet** to go to the relevant section of code.

The final step in returning data is to implement a `DSISimpleResultSet`. The sample contains an implementation called `ULResultSet` which returns a hard-coded set of people's names.

A `DSISimpleResultSet` implementation contains the data result from a query execution, which the calling framework will use to access each row and column of data.

Your implementation should maintain a handle to a cursor within the SQL enabled data source, and delegate calls to the data source to move to the next row when the `MoveToNextRow()` method is called.

In the UltraLight sample connector, `ULResultSet::MoveToNextRow()` simply increments an row iterator. In your implementation, replace this section with code that delegates this functionality to the data source.

The `RetrieveData()` method is where column data is retrieved. Modify this method to extract data from the data source. For more information about data retrieval, see Data Retrieval on page 51.

Summary of Day Four

You can now execute queries and retrieve data from your data store. You can use any ODBC-enabled application to execute queries and see the results returned from your data store.

Day Five

Day Five instructions explain how to rebrand your custom ODBC connector.

Rebrand Error Messages

Error messages sent by the connector are visible to applications and customers. In the UltraLight sample connector, error messages are branded with `UltraLight`, `UL`, and `Simba`. This section explains how to rebrand the error messages to reflect the custom connector name and the company name.

All the error messages used within the DSI implementation are stored in a file called `ULMessages.xml`.

To configure error messages:

1. Rename the `ULMessages.xml` file to reflect the name of your company or your custom ODBC connector.
2. Double click the **TODO #14 Register the ULMessages.xml file for handling by DSIMessageSource** message to jump to the relevant section of code.
3. Update the line associated with the **TODO** to match the new name of the `ULMessages.xml` file.
4. Open the `ULMessages.xml` file and change all instances of the following items:
    - Change the letters `UL` to an appropriate two-letter abbreviation.
    - Change the word `UltraLight` to an appropriate name for your custom connector.
5. For each exception thrown within the custom DSI implementation, change the parameters to match your custom connector name. This rebrands the error messages to reflect the name of your connector.
6. Double click the **TODO #15 Set the vendor name, which will be prepended to error messages** message to go to the relevant section of code.
7. The vendor name is prepended to all error messages that are visible to applications. As explained in the code comments, change the vendor name from `Simba` to an appropriate name for your company.

How can I update the vendor name in the Tableau Datasource Connection (TDC) file?

A TDC file contains configuration information that will be applied to any Tableau connection that matches the database vendor name and connector name described in the TDC file.

To set the vendor name for your custom ODBC connector:

1. Ensure you have set your vendor name as described in **TODO #15**.
2. In the class that extends `DSIConnection` (`ULConnection` in our sample UltraLight Connector), set the property DSI_CONN_DBMS_NAME to your vendor name.

   Example:
   ```
   SetProperty(DSI_CONN_DBMS_NAME,
   AttributeData::MakeNewWStringAttributeData
   ("YourVendorName"));
   ```

   By default, the value of the DSI_CONN_DBMS_NAME property is `TEXT`.
3. Set the vendor name in the TDC file by following the instructions on Tableau's website.

These steps allow Tableau to match the vendor name in the TDC file with the associated `SQLGetInfo()` property it queries the connector for.

Rebrand the Custom ODBC Connector

All the TODOs in the UltraLight sample connector project are finished, and the custom connector is rebranded and retrieving data from your data store. To complete the custom connector, add the following functionality:

1. Rename all files and classes in the project to have the two-letter abbreviation chosen as part of **TODO #14**.
2. Create a connector configuration dialog. This dialog is presented to the user when the ODBC Data Source Administrator is used to create a new ODBC DSN or configure an existing one. The UltraLight connector project contains an example ODBC configuration dialog under the `Setup` folder in the UltraLight connector project.
3. To see the connector configuration dialog that was created in the previous step, run the ODBC Data Source Administrator, open the Control Panel, select Administrative Tools, and then select Data Sources (ODBC). If the Control Panel is set to view by category, then Administrative Tools is located under System and Security.

Important:

If you are using 64-bit Windows with 32-bit applications, you must use the 32-bit ODBC Data Source Administrator. You cannot access the 32-bit ODBC Data Source Administrator from the Start menu or control panel in 64-bit Windows. To access a 32-bit ODBC Data Source Administrator from a 64-bit machine, run `C:\WINDOWS\SysWOW64\odbcad32.exe`.

For more information, see Bitness and the Windows Registry on page 50 and 32-bit vs 64-bit ODBC Data Source Administrator on page 49.

Conclusion

You have written a custom ODBC connector that can be used by ODBC-enabled applications to query and retrieve data from a custom data store. The custom ODBC connector is renamed and rebranded for your company and product.

Reference

This section contains more information that you may find useful when developing your sample ODBC driver.

32-bit vs 64-bit ODBC Data Source Administrator

On a 64-bit Windows machine, you can execute both 64-bit and 32-bit applications. Many applications are available in 32-bit versions only, and running 32-bit applications on 64-bit operating systems is common.

However, in a single running process, all of the code must be either 32-bit or 64-bit. A connector must have the same bitness as the application that loads it.

Important:

- 64-bit applications can only load 64-bit connectors and 32-bit applications can only load 32-bit connectors.
- In a single running process, all of the code must be either 64-bit or 32-bit.

Note:

- Microsoft Excel is available in 32-bit and 64-bit versions.

Using the Correct ODBC Data Source Administrator

When using 64-bit Windows, it is very important that you configure 32-bit connectors with the 32-bit ODBC data source administrator, and 64-bit connectors with the 64-bit ODBC Data Source Administrator. This can cause confusion, where what appears to be a perfectly configured ODBC DSN does not work because it is loading the wrong kind of connector.

On a 64-bit Windows machine, use the Control Panel to launch the 64-bit ODBC data source administrator.

To launch the 32-bit ODBC data source administrator, run the following:

`C:\WINDOWS\SysWOW64\odbcad32.exe.`

Tip:

Create a shortcut to the 32-bit ODBC Data Source Administrator on your Desktop or Start menu if you configure 32-bit data sources frequently.

Bitness and the Windows Registry

32-bit applications can run on 64-bit machines, but they cannot use 64-bit ODBC connectors. A 64-bit application must use a 64-bit ODBC connector, and a 32-bit application must use a 32-bit ODBC connector.

On machines running 64-bit Windows operating systems, system-wide information about 64-bit ODBC connectors is stored in **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC**, and system-wide information about 32-bit ODBC connectors is stored in **HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC**.

On machines running 32-bit Windows operating systems, system-wide information about ODBC connectors is stored in **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC**. This is the same location as 64-bit applications running on 64-bit machines.

Note:

32-bit Windows operating systems cannot run 64-bit applications or connectors.

The ODBC.INI Key

The ODBC Data Source Administrator uses information in the following key to connect a connector to a database:

- **HKEY_LOCAL_MACHINE/SOFTWARE/ ODBC/ODBC.INI** for 64-bit applications on 64-bit machines, or 32-bit applications on 32-bit machines.
- **HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ ODBC.INI** for 32-bit applications on 64-bit machines.

This key contains a key for each Data Source Name (DSN). For more information about this key, see https://msdn.microsoft.com/en-us/library/ms715391(v=vs.85).aspx.

The ODBC Data Sources key

The **ODBC.INI** key also contains a key named **ODBC Data Sources** that lists the data sources. The values for the data sources must match the name of each DSN key. For more information about this key, see https://msdn.microsoft.com/en-us/library/ms709335(v=vs.85).aspx.

For more information on Windows Registry entries for data sources, see https://msdn.microsoft.com/en-us/library/ms712603%28v=vs.85%29.aspx.

The ODBCINST.INI Key

The ODBC Data Source Administrator uses information in the following key to define each connector's name and setup location:

- **HKEY_LOCAL_MACHINE/SOFTWARE/ODBC/ODBCINST.INI** for 64-bit applications on 64-bit machines, or 32-bit applications on 32-bit machines
- **HKEY_LOCAL_ MACHINE/SOFTWARE/WOW6432NODE/ODBC/ODBCINST.INI** for 32-bit applications on 64-bit machines

This key contains a connector specification key for each connector. For more information about connector specification keys, see https://msdn.microsoft.com/en-us/library/ms715391(v=vs.85).aspx.

The ODBC Connectors Key

The **ODBCINST.INI** key also contains a key named **ODBC Drivers** that lists the installed connectors.

For more information about this key, see https://msdn.microsoft.com/en-us/library/ms714818%28v=vs.85%29.aspx.

Data Retrieval

In the Data Store Interface (DSI), the following methods perform the actual task of retrieving data from your data store:

- Each `MetadataSource` implementation of `GetMetadata()`
- `DSISimpleResultSet::RetrieveData()`

Both methods provide a way to uniquely identify a column within the current row. For `MetadataSource`, the Simba SDK passes in a unique column tag (see `DSIOutputMetadataColumnTag`). For `ULResultSet`, the Simba SDK will pass in the column index.

In addition, both methods accept the following three parameters:

- `in_data`

  The SQLData into which you must copy the value of your cell. This class is a wrapper around a buffer managed by the Simba SDK. To access the buffer, you call its `GetBuffer()` method. The data you copy into the buffer must be formatted as a SQL Type (see https://msdn.microsoft.com/en-us/library/ms710150%28VS.85%29.aspx for a list of data types and definitions). Therefore, if your data is not stored as SQL Types, you will need to write code to

convert from your native format.

The type of this parameter is governed by the metadata for the column that is returned by the class. Thus, if you set the SQL Type of column 1 in DSISimpleResultSet::InitializeColumns() to SQL_INTEGER, then when DSISimpleResultSet::RetrieveData() is called for column 1, you will be passed a SQLData that wraps a simba_int32 (or simba_uint32 if unsigned) data type. For MetadataSource, the type is associated with the column tag (see DSIOutputMetadataColumnTag.h).

Example:

If `SqlData` was of type `SQL_INTEGER`:

```
simba_int32 value = 5;
//This is one way
memcpy(in_data->GetBuffer(), &value, sizeof(simba_int32));
// This is another way; both work equally well
*reinterpret_cast<simba_int32*>(in_data->GetBuffer()) = 5;
```

When working with variable length data, for example character or binary data, you must call `SetLength()` before calling `GetBuffer()`. Not doing so may result in a heap violation. See `ULTypeUtilities.h` for an example on how to handle character or binary data.

- `in_offset`

  Character, wide character and binary data types can be retrieved in parts. This value specifies where, in the current column, the value should be copied from. The value is usually 0.

- `in_maxSize`

  The maximum size (in bytes) that can be copied into the `in_data` parameter. For character or binary data, copying data that is greater than this size can result in a data truncation warning or a heap violation.

SqlData Types

`SqlData` objects represent the SQL types and encapsulate the data in a buffer. To get the underlying SQL type that a `SqlData` object represents, use `GetMetadata()->GetSqlType()`. This retrieves the associated `SQL_*` type.

For information on how SQL types map to C++ types, see *SQL Data Types* in Developing Connectors for SQL-capable Data Stores

NULL Values

To represent a null value, directly set the `SqlData` object as null:

```
in_data->SetNull(true);
```

Server Configuration

Your custom ODBC connector can be recompiled as a server and deployed in a client-server configuration. The connection settings for the connector are normally retrieved directly from the ODBC DSN. However, when the connector is a server, the settings cannot be retrieved directly because the DSN refers to the client instead of a specific connector. Also, to enforce security, clients do not have control over server-specific settings.

For information about making a connection to a connector that is compiled and built as a server, see the SimbaClient/Server Developer Guide.

Install the Evaluation License

You can use Simba SDK for 30 days after installing the evaluation license. The evaluation license is emailed to the person who registered the product.

Typically, you use Simba SDK to create your custom ODBC connector, then use a test ODBC-enabled application to retrieve data using the connector. Install the license file to the appropriate location for the type of ODBC-enabled application you are running.

Install the Evaluation License on Windows

To license Simba SDK for applications running under a non-SYSTEM user account:

- Save the license file in the `%USERPROFILE%` folder, for example:

  ```
  C:\Users\<USER_ID>\SimbaEngineSDK.lic
  ```

  Where *<USER_ID>* is the ID of the user running the application.

To license Simba SDK for applications running under a SYSTEM user account:

- For 32-bit applications on 32-bit machines, or 64-bit applications on 64-bit machines, save the license file as follows:

  ```
  C:\Windows\System32\config\systemprofile\SimbaEngineSDK.lic
  ```

- Or, for 32-bit applications running on 64-bit machines, save the license file as follows:

```
C:\Windows\SysWOW64\config\systemprofile\SimbaEngineSDK.li
c
```

Contact Us

For more information or help using this product, please contact our Technical Support staff. We welcome your questions, comments, and feature requests.

Note:

To help us assist you, prior to contacting Technical Support please prepare a detailed summary of the Simba SDK version and development platform that you are using.

You can contact Technical Support via the Magnitude Support Community at www.magnitude.com.

You can also follow us on Twitter @SimbaTech and @Mag_SW.

Third-Party Trademarks

Simba, the Simba logo, Simba SDK, and Simba Technologies are registered trademarks of Simba Technologies Inc. in Canada, United States and/or other countries. All other trademarks and/or servicemarks are the property of their respective owners.

Kerberos is a trademark of the Massachusetts Institute of Technology (MIT).

Linux is the registered trademark of Linus Torvalds in Canada, United States and/or other countries.

Mac and macOS are trademarks or registered trademarks of Apple, Inc. or its subsidiaries in Canada, United States and/or other countries.

Microsoft SQL Server, SQL Server, Microsoft, MSDN, Windows, Windows Azure, Windows Server, Windows Vista, and the Windows start button are trademarks or registered trademarks of Microsoft Corporation or its subsidiaries in Canada, United States and/or other countries.

Red Hat, Red Hat Enterprise Linux, and CentOS are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in Canada, United States and/or other countries.

Solaris is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

SUSE is a trademark or registered trademark of SUSE LLC or its subsidiaries in Canada, United States and/or other countries.

Ubuntu is a trademark or registered trademark of Canonical Ltd. or its subsidiaries in Canada, United States and/or other countries.

All other trademarks are trademarks of their respective owners.