



Magnitude Simba SDK

**Build a C++ ODBC Driver for SQL-Capable Data Sources in
5 Days (MAC OS X)**

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 Simba SDK	14
Build the Sample ODBC Connector	16
Configure the Connector and Data Source	18
Connect to the Data Source	19
Set up a Custom ODBC Connector Project	21
Configure Your Custom Connector and Data Source	22
Debug Your Custom Connector	24
Enable Logging	26
Day Two	28
Set the Configuration Branding	28
Set Connector Properties	28
Set Logging Details	29
Check Connection Settings	30
Customize the DriverPrompt Dialog	32
Establish a Connection	33
Day Three	34
Create and Return Metadata Sources	34
Day Four	40
Prepare and Execute a Query	40
Day Five	43
Rebrand Error Messages	43
Rebrand the Custom ODBC Connector	44
Reference	45
Driver Managers	45

Locating the Configuration Files	46
Data Retrieval	48
Server Configuration	50
Install the Evaluation License	50
Troubleshooting	50
Contact Us	52
Third-Party Trademarks	53

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

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.

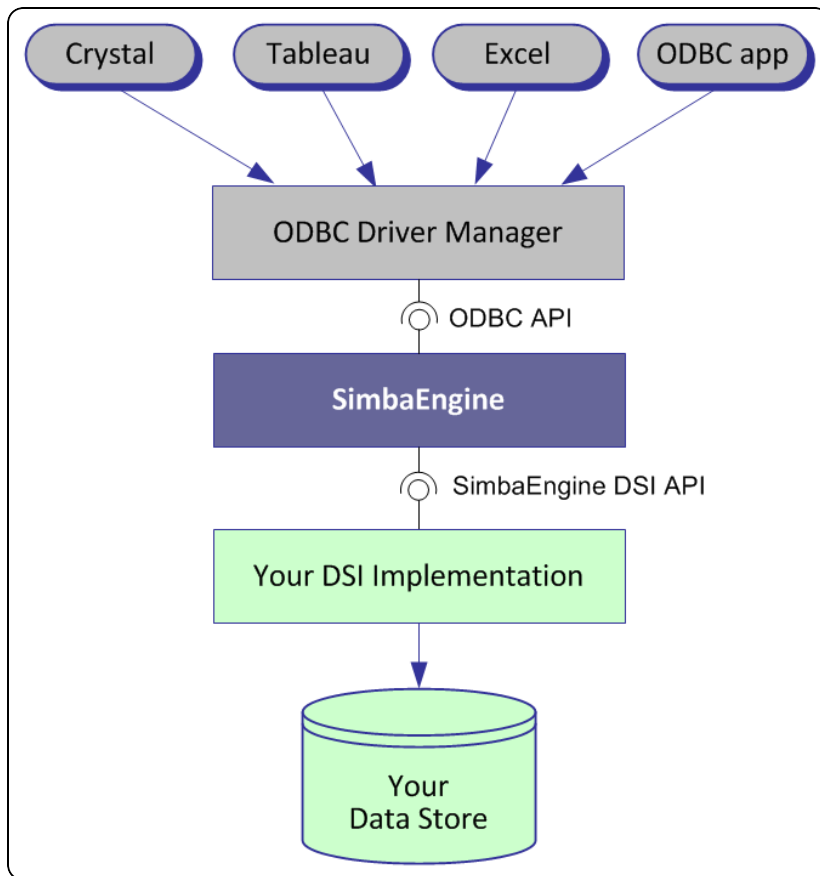
For a brief description of the ODBC standard, see <http://www.simba.com/resources/data-access-standards-library#!odbc>.

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](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.

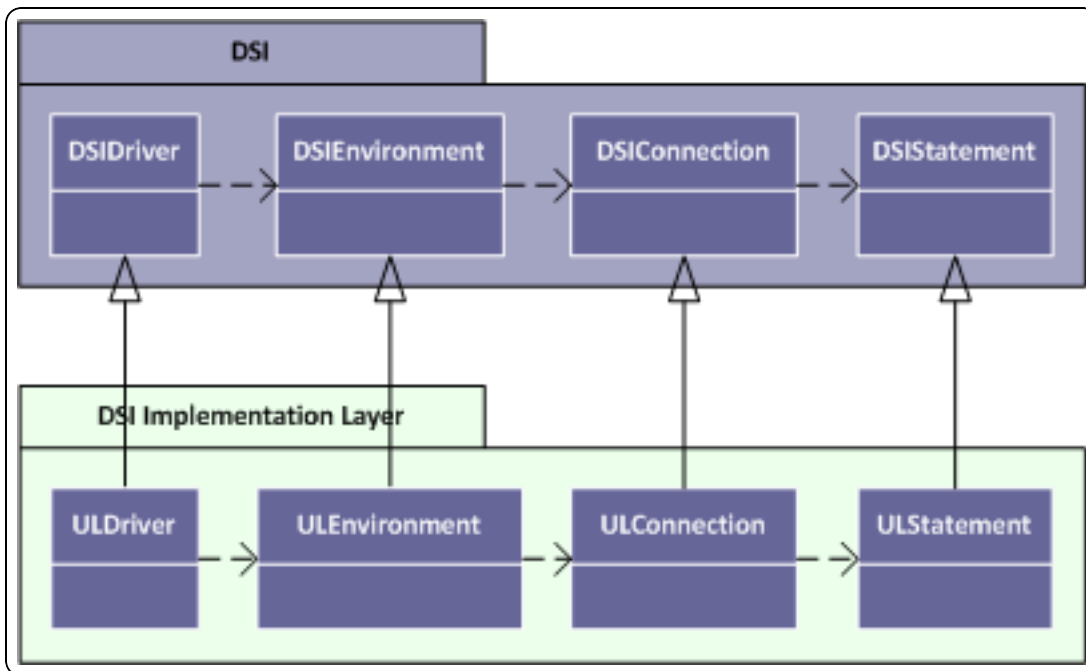
About the UltraLight Sample Connector

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.

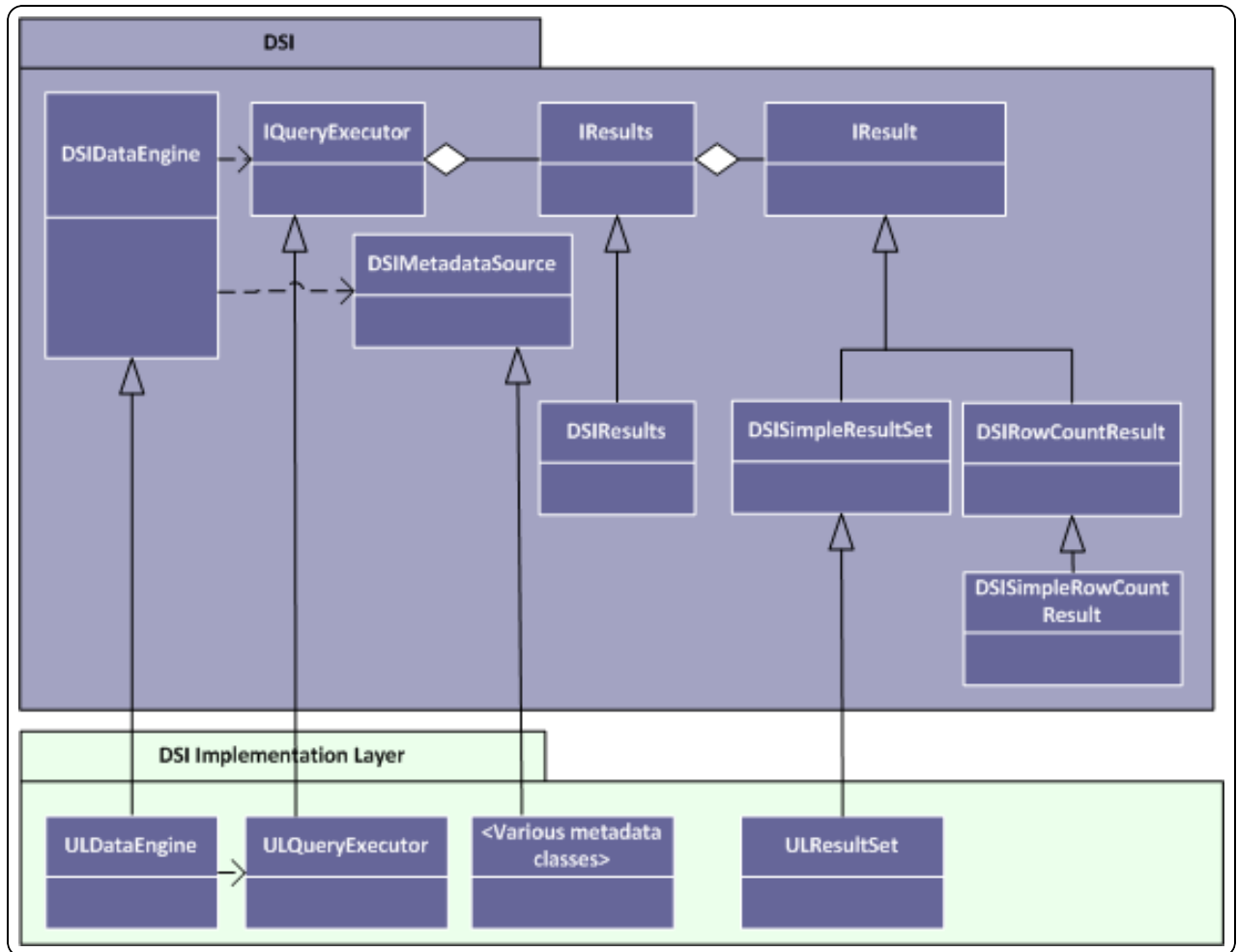
Implementation begins with the creation of a `DSIDriver` class which is responsible for constructing a `DSIEnvironment`. `DSIEnvironment` is used to construct a connection object (`DSIConnection` implementation) which is then used for constructing statements (`DSIStatement` implementations).

This concept is summarized in the figure below:

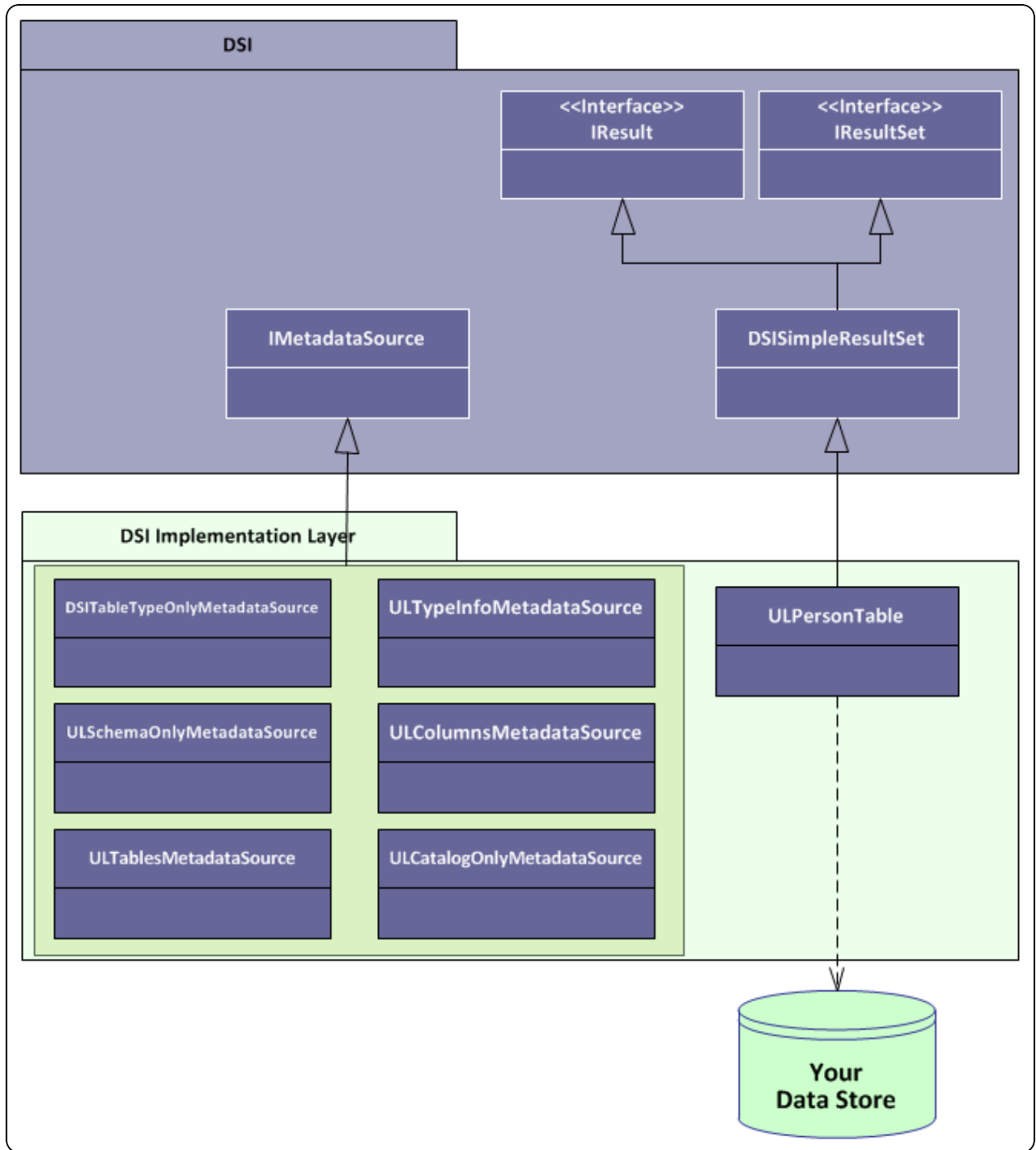


The `DSIStatement` implementation is responsible for creating a `DSIDataEngine` object, which then creates `IQueryExecutor` objects to execute queries and hold results (`IResults`), and `DSIMetadataSource` objects to return metadata information.

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 Simba SDK

The Simba SDK for macOS platforms includes an installer to help you install and configure the product correctly. This installer removes any previous installations, installs the product, and configures the DSN and other configuration files for the sample connectors.

Important:

If a previous installation of the Simba SDK with the same product and compiler version exists on your machine, the installer removes it. We recommend that you back up any previous work before reinstalling the product.

Note:

If the ODBC configuration files `odbc.ini` and `odbcinst.ini` exist on your machine, the installer modifies them to add information for the sample connectors. Both the system-wide configuration files under `/Library/ODBC` and the user-specific files under `~/Library/ODBC` are modified. ODBC configuration files in other locations are not modified.

If these files do not exist on your machine, the installer creates them.

To install the Simba SDK:

The Simba SDK is distributed as a `.dmg` file inside a `.tar.gz` file.

1. Ensure you have global administrator privileges on your machine.
2. Uninstall any previous versions of the Simba SDK.

3. Ensure the evaluation license is installed. See [Install the Evaluation License](#).
4. Copy the `SimbaEngineSDK[BUILD].tar.gz` file to the Simba SDK installation directory, where `[BUILD]` is the build number and platform.

Ensure you copy the correct version of the Simba SDK for your platform. To determine your machine version, type `uname -m` at the command prompt.

5. Uncompress the file by typing the following:

```
gunzip SimbaEngineSDK[BUILD].tar.gz
```

6. Extract the `.tar` file by typing the following:

```
tar -xvf SimbaEngineSDK[BUILD].tar
```

7. Double-click the `.dmg` file, then double-click the `.pkg` file.

The installer launches.

8. In the Introduction window, select **Continue**.
9. In the Read Me window, read the release notes then select **Continue**.
10. In the License window, read the license agreement then select **Continue**.
11. In the resulting window, if you agree with the license conditions, select **Agree**.
12. In the Installation Type window, if you agree with the install location, select **Install**. Otherwise, select **Change Install Location** and select a different disk.

Note:

- The installer always installs the Simba SDK to `/Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK`, but you can choose which disk.
- On macOS platforms, Microsoft Excel 2016 only load connectors from `/Library` and `/Applications`.

13. Enter the username and password to complete the installation. The user must have global administrator privileges.
14. In the final installation window, select **Close**.

The Simba SDK is installed to the location `/Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK`.

The following system-wide environment variables are set using the `plist` file `/Library/LaunchAgents/setenv.SIMBAENGINESDK.plist`:

- **SIMBAENGINE_DIR**=`/Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK/10.2/DataAccessComponents`

- **SIMBAENGINE_THIRDPARTY_DIR**=/Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK/10.2/DataAccessComponents/ThirdParty

[Install the Evaluation License](#)

[Driver Managers](#)

Build the Sample ODBC Connector

You can use the sample makefile to build the UltraLight connector. The sample makefile automatically detects the required settings based on your operating system, machine bitness, and compiler. For more information about different makefile options, see *Compiling Your Connector* in the guide [Developing Connectors for SQL-capable Data Stores](#).

To build the Simba SDK UltraLight sample connector:

The sample connectors included with the Simba SDK are installed in the folder /Library/Simba_XCode[XCode_VERSION]/SimbaEngineSDK/10.2/Examples. The sample connectors include sample makefiles.

In the following instructions, replace [INSTALL_DIR] with the Simba SDK installation directory, for example /Library/Simba_XCode7.

1. Change to the following directory:
[INSTALL_DIR]/SimbaEngineSDK/10.2/Examples/Source/UltraLight/Source
2. Type `./mk.sh MODE=debug` to run the makefile for the debug target.

Important:

Do not use the makefile directly. Use the `mk.sh` script instead.

3. By default, the makefile detects the latest version of XCode on your machine. Optionally, you can specify a different version of XCode using the environment variable **DEVELOPER_DIR**. For an example, see *Compiling your Connector at Developing Connectors for SQL-capable Data Stores*.

The resulting library, `libUltraLight<BITNESS>.dylib`, is put in the following directory:


```
[INSTALL_  
DIR]/SimbaEngineSDK/  
10.2  
/Examples/Source/  
UltraLight/Bin/<BUILD>/<RELEASE|DEBUG><BITNESS>.
```

Where **<BUILD>** is a combination of your operating system, machine bitness, and compiler, **<RELEASE|DEBUG>** is either release or debug, and **<BITNESS>** is 32, 64, or 3264.

Example Build Location Using XCode 7

```
/Library/Simba_  
XCode7/SimbaEngineSDK/  
10.2/Examples/Source/UltraLight/Bin/Darwin_x86_  
Xcode7/debug64/libUltraLight64.dylib
```

Extracting 32- or 64-bit libraries from a universal library

You can use the `lipo` command to extract 32- or 64-bit libraries from a universal library.

To extract i386 or x86_64 libraries from the universal library:

1. Navigate to the following directory:

```
[INSTALL_  
DIR]/SimbaEngineSDK/  
10.2/Examples/Source/UltraLight/Bin/Darwin_x86_xcode  
[XCode_VERSION]/<RELEASE|DEBUG><BITNESS>
```

2. To display a list of the architectures found in the UltraLight sample connector universal library created in the previous step, type the following:

```
lipo -info libUltraLight3264.dylib
```

The following information appears:

```
Architectures in the fat file: libUltraLight3264.dylib  
are: i386 x86_64
```

3. To extract i386 libraries from the universal library, type:

```
lipo libUltraLight3264.dylib -extract i386 -output  
libUltraLight32.dylib
```

4. To extract x86_64 libraries, type:

```
lipo libUltraLight3264.dylib -extract x86_64 -output  
libUltraLight64.dylib
```

The library is extracted.

Compiling Your Connector in the guide [Developing Connectors for SQL-capable Data Stores](#)

Configure the Connector and Data Source

On macOS platforms, the Simba SDK supports the iODBC driver manager. The driver manager uses configuration files to locate and load ODBC connectors. The `odbc.ini` file defines ODBC data sources, or DSNs, and the `odbcinst.ini` file defines ODBC connectors.

If the ODBC configuration files `odbc.ini` and `odbcinst.ini` exist on your machine, the installer modifies them to add information for the sample connectors. Both the system-wide configuration files under `/Library/ODBC` and the user-specific files under `~/Library/ODBC` are modified. ODBC configuration files in other locations are not modified. If these files do not exist on your machine, the installer creates them.

To configure the UltraLight connector and data source:

1. In the `/Library/ODBC` directory, open the `odbc.ini` configuration file in a text editor.
2. Make sure there is an entry in the `[ODBC Data Sources]` section that defines the data source name (DSN) of the UltraLight sample connector.

Example:

```
[ODBC Data Sources]
UltraLightDSII=UltraLightDSIIDriver
```

3. Make sure there is a section with a name that matches the data source name (DSN).

Example:Using XCode 7

```
[UltraLightDSII]
Description=64-bit UltraLight DSII
Driver=/Library/Simba_
XCode7/SimbaEngineSDK/
10.2/Examples/Source/UltraLight/Bin/Darwin_x86_
Xcode7/debug64/libUltraLight64.dylib
```

4. Save and close the file.
5. Open the `odbcinst.ini` configuration file in a text editor.
6. Add a new entry to the `[ODBC Drivers]` section.

Example:

```
[ODBC Drivers]
UltraLightDSIIDriver=Installed
```

7. Add a new section with a name that matches the new connector name.

Example: Using XCode 7

```
[UltraLightDSIIDriver]
Driver=/Library/Simba_
XCode7/SimbaEngineSDK/
10.2/Examples/Source/UltraLight/Bin/Darwin_x86_
Xcode7/debug64/libUltraLight64.dylib
```

8. Save and close the file

Your custom ODBC connector and data source are configured.

[Locating the Configuration Files](#)

Connect to the Data Source

In order for the UltraLight sample connector to connect to the sample database successfully, the **DYLD_LIBRARY_PATH** environment variable must include references to the OpenSSL and ICU libraries.

Note:

- You must have a driver manager installed. See [Driver Managers](#).
- You must use a 64-bit driver manager with a 64-bit connector, or a 32-bit driver manager with a 32-bit connector.

To add the OpenSSL library and the ICU library to the library path:

1. Locate the path to the correct version of the OpenSSL library for your platform, compiler, and machine bitness. For example:

```
/Library/Simba_
XCode6/SimbaEngineSDK/
10.2/DataAccessComponents/ThirdParty/openssl/1.0.1/Darwin_
x86_Xcode6/release3264/lib
```

2. Add the OpenSSL path to the **DYLD_LIBRARY_PATH** environment variable. For example:

```
export DYLD_LIBRARY_PATH=DYLD_LIBRARY_PATH:[Path to
OpenSSL]
```

3. Locate the path to the correct version of the ICU library for your platform, compiler, and machine bitness. For example:

```
/Library/Simba_  
XCode6/SimbaEngineSDK/  
10.2/DataAccessComponents/ThirdParty/icu/53.1.x/Darwin_  
x86_Xcode6/release3264/lib
```

4. Add the ICU path to the DYLD_LIBRARY_PATH environment variable. For example:

```
export DYLD_LIBRARY_PATH=DYLD_LIBRARY_PATH:[Path to ICU]
```

OpenSSL is used by SimbaClient for ODBC. Your custom ODBC connector may not require this library.

To test connecting the UltraLight sample connector to the sample data source:

This procedure uses the `iodbctest` utility that is included with the iODBC driver manager. For help, see [Troubleshooting](#).

1. At the command prompt, type `iodbctest`.
2. At the prompt that says “Enter ODBC connect string”, type `?` to show the list of DSNs and Drivers.

The list contains UltraLightDSII DSN.

3. To connect to your data source, type: `DSN=UltraLightDSII;UID=a;PWD=b`

The prompt `SQL>` appears.

4. Type a SQL command to query the database. For example:

```
SELECT * FROM ULResultSet
```

The SQL results are returned.

5. To quit `iodbctest`, type `quit` at the prompt.

Note:

You can use other ODBC-enabled applications to test the sample connector. Note that Microsoft Excel 2016 will only load connectors from `/Library` and `/Applications`.

Once the UltraLight sample connector project is built, it can be used to create a custom connector project.

[Troubleshooting](#)

Set up a Custom ODBC Connector Project

Once the UltraLight project has been built and tested, you can create a custom project for your 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 create a custom ODBC connector project based on the UltraLight sample connector:

1. Ensure you are working in a window or shell where the **SIMBAENGINE_DIR** and **SIMBAENGINE_THIRDPARTY_DIR** environment variables are set, as explained in [Build the Sample ODBC Connector](#).
2. Copy the `UltraLight` directory to create a new top-level directory for your custom ODBC connector project. Be sure to copy hidden files and symlinks too, for example:

```
mkdir [INSTALL_  
DIR]/SimbaEngineSDK/10.2/Examples/Source/MyUltraLight  
cp -a [INSTALL_  
DIR]/SimbaEngineSDK/10.2/Examples/Source/UltraLight/.  
[INSTALL_  
DIR]/SimbaEngineSDK/10.2/Examples/Source/MyUltraLight
```

where `[INSTALL_DIR]` is the Simba SDK installation directory, for example `/Library/Simba_XCode7`.

This new directory is referred to as the `[Project]` directory in the following steps.

3. In the `[Project]/Source` directory, open the `GNUmakefile` in a text editor.
4. Replace the `target.driver` target name with the name of your custom connector.

Example:

Replace this line: `target.driver = libUltraLight${BITS}.${SO}`

With this line: `target.driver = libMyUltraLight${BITS}.${SO}`

5. Save and close the file.
6. In the `[Project]/Source` directory, run the following command to build your custom ODBC connector:

```
./mk.sh MODE=debug
```

Your custom ODBC connector project is built.

Compiling Your Connector in the guide [Developing Connectors for SQL-capable Data Stores](#)

[Troubleshooting](#)

Configure Your Custom Connector and Data Source

On macOS platforms, the Simba SDK supports the iODBC driver manager . This driver manager uses configuration files to define and configure ODBC data sources and connectors. The `odbc.ini` file is used to define ODBC data sources and the `odbcinst.ini` file is used to define ODBC connectors. Connector-specific information , such as log file location, is configured in the `.simba.UltraLight.ini` file.

To configure the `odbc.ini` file:

1. Open the `/Library/ODBC/odbc.ini` configuration file in a text editor.
2. Make sure there is an entry in the `[ODBC Data Sources]` section that defines the data source name (DSN).

Example:

```
[ODBC Data Sources]
MyUltraLightDSII=MyUltraLightDSIIDriver
```

3. Make sure there is a section with a name that matches the data source name (DSN).

Example:

```
[MyUltraLightDSII]
Description=Sample SimbaEngine UltraLight DSII
```

```
Driver=/Library/Simba_  
XCode7/SimbaEngineSDK/  
10.2/Examples/Source/MyUltraLight/Bin/Darwin_x86_  
Xcode7/debug64/libMyUltraLight64.dylib  
Locale=en-US
```

4. Save and close the file.

To configure the `odbcinst.ini` file:

1. Open the `/Library/ODBC/odbcinst.ini` configuration file in a text editor.
2. Add a new entry to the `[ODBC Drivers]` section. For example:

```
[ODBC Drivers]  
MyUltraLightDSIIDriver=Installed
```

3. Add a new section with a name that matches the new connector name.

Example:

```
[MyUltraLightDSIIDriver]  
Driver=/Library/Simba_  
XCode7/SimbaEngineSDK/  
10.2/Examples/Source/MyUltraLight/Bin/Darwin_x86_  
Xcode7/debug64/libMyUltraLight64.dylib
```

4. Save and close the file.

To configure the `.simba.UltraLight.ini` file:

1. Copy the sample `.simba.UltraLight.ini` file to the user's home directory. Note that the sample `.simba.UltraLight.ini` file is hidden.
2. Open the `~/simba.UltraLight.ini` configuration file in a text editor.
3. Replace every instance of `[INSTALLDIR]` with the installation location of the Simba SDK.
4. Set the `DriverManagerEncoding` setting to `UTF-32`.

Note:

- This step is optional, because the Simba SDK automatically detects the type and version of the driver manager. Set the `DriverManagerEncoding` only if you want to override the value that is automatically detected.
- If you are unsure where the driver manager is installed, contact your system administrator or see [Driver Managers](#) for more information.

5. Edit the `ErrorMessagesPath` setting to replace `[INSTALLDIR]` with your install directory.
6. Set the `ODBCInstLib` to the absolute path of the `ODBCInst` library for the Driver Manager that you are using.

Note:

This step is optional, because the Simba SDK automatically detects the type and version of the `ODBCInst` library. Set this value only if you want to override the value that is automatically detected.

The `ODBCInst` library is a part of the driver manager but is used by the connector to read values from the `odbc.ini` file. The value of this key is the absolute path of the `ODBCInst` library. For the iODBC Driver Manager this would be `<driver manager dir>/lib/libiodbcinst.dylib` (notice the 'i' after the lib).

7. Save the file.

The custom connector and data source are configured.

[Configure the Connector and Data Source](#)

[Locating the Configuration Files](#)

Debug Your Custom Connector

You can use a debugger to step through the custom connector code and gain a better understanding of the connector's functionality. This section explains how to use the `iodbctest` application to connect to the custom connector, then use the LLDB debugger to step through the connector code.

To debug the custom connector code:

1. Follow the instructions in [Connect to the Data Source](#) to use `iodbctest` to connect to the custom connector. Use the name of your custom ODBC connector instead of the UltraLight Connector.
2. To quit `iodbctest`, type `quit` at the command prompt.
3. To start the debugger, type `lldb iodbctest`.

4. Type the following:

```
b Simba::DSI::DSIDriverFactory
```

This sets a breakpoint at the `DSIDriverFactory()` function in the `Main_Unix.cpp` file. This is a good breakpoint to start with, because this function runs as soon as the driver manager loads the ODBC connector.

Note:

LLDB may display a message that the breakpoint cannot be resolved to a location because the connector is not loaded yet. This will be resolved in a following step when you run the connector.

5. To set a different breakpoint, view the source code the following directory:

```
[INSTALL_DIR]/SimbaEngineSDK/10.2/Examples/Source/SimbaSDK/Source/
```

6. To load and run the connector until the breakpoint is encountered, type:

```
run
DSN=MyUltraLightDSII;UID=<YourUserName>;PWD=<YourPassword>
```

The program runs until the breakpoint is encountered.

Note:

When using the `lldb` debugger with an application, the ODBC connector is not loaded until the application is running and a connection is made. This means that breakpoints can be set either before or after the connector is loaded, depending on which breakpoint you want to hit.

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

[ODBC Troubleshooting: How to Enable Driver-manager Tracing](#)

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. Open the `.simba.ultralight.ini` file in the user's home directory.

Note:

Your custom ODBC connector is not yet completely rebranded, so configuration information is read from the `.simba.ultralight.ini` file. The name of this file is set using `#define DRIVER_LINUX_BRANDING` in the `Main_Unix.cpp` file.

2. Set the `LogLevel` to `6` for trace level, or another level if you prefer.
3. Set the `LogPath` to the directory to use for log files.

Example:

```
LogLevel=6
LogPath=/usr/tmp/myultralight/logs
```

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"
```

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.

<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.

Set the Configuration Branding

The `DSIDriverFactory()` implementation in `Main_Unix.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 shared object.

To construct the connector singleton:

1. In your custom ODBC connector project, open the file `Main_Unix.cpp`.
2. Navigate to the line **TODO #1: Construct connector singleton**.
3. Look at the `DSIDriverFactory()` implementation, and locate the following line of code:

```
SimbaSettingReader::SetConfigurationBranding(DRIVER_LINUX_BRANDING);
```
4. In the default implementation, `DRIVER_LINUX_BRANDING` defines the string `"simba.UltraLight.ini"`. This is the name of the `.ini` file that specifies the connector settings.
5. Change this string to the name of a configuration file reflecting the name of your connector or company.
6. Save the `Main_Unix.cpp` file.
7. Update the name of the `"simba.UltraLight.ini"` file to match `DRIVER_LINUX_BRANDING`.

Set Connector Properties

To set connector properties:

1. Open the file `ULDriver.cpp` file and navigate to the line **TODO #2 Set the connector properties**.
2. 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.
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. Open the file `ULDriver.cpp` and navigate to the line **TODO #3 Set the connector-wide logging details**.
2. Change the connector log's file name.
3. Open the file `ULConnection.cpp` and navigate to the line **TODO #4 Set the connection-wide logging details**.
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. Open the file `ULConnection.cpp` and navigate to the line **TODO #5 Check Connection Settings**.

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 can call `ULConnection::PromptDialog()` to display a dialog box, prompting the user for information about the connection.

Open the file `ULConnection.cpp` and navigate to the line **TODO #6 Customize DriverPrompt Dialog**. Update this method with the required and optional settings for your custom ODBC connector.

Note:

The UltraLight sample connector code can be compiled and run on both Windows and Linux/Unix/macOS platforms. While this method creates a dialog on Windows, on Linux/Unix/macOS there is no sample dialog. Instead, the window handle and prompt enum are ignored, while the connection settings parameter is populated with empty values for the user ID and password fields:

```
(io_connectionSettings)[UL_UID_KEY] = Variant(simba_wstring(""));
(io_connectionSettings)[UL_PWD_KEY] = Variant(simba_wstring(""));
```

You must add code to retrieve these values from a location such as a dialog box or a configuration file.

The `ULConnection::PromptDialog()` 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 are not supplied by the user.
- `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 to request just the required settings, or both the optional and the required settings.

The return value for this method indicates whether the user completed the process by clicking OK on the dialog box (return true), or whether the user aborted the process by

clicking CANCEL on the dialog box (return false). `PromptDialog()` is called multiple times if the user clicks OK but does not enter all the required input.

Before `PromptDialog()` is called, `UpdateConnectionSettings()` is called. This method checks if any required settings are not provided, then updates `in_connResponseMap` accordingly.

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. Open the file `ULConnection.cpp` and navigate to the line **TODO #7 Establish A Connection**.
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 error. 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.

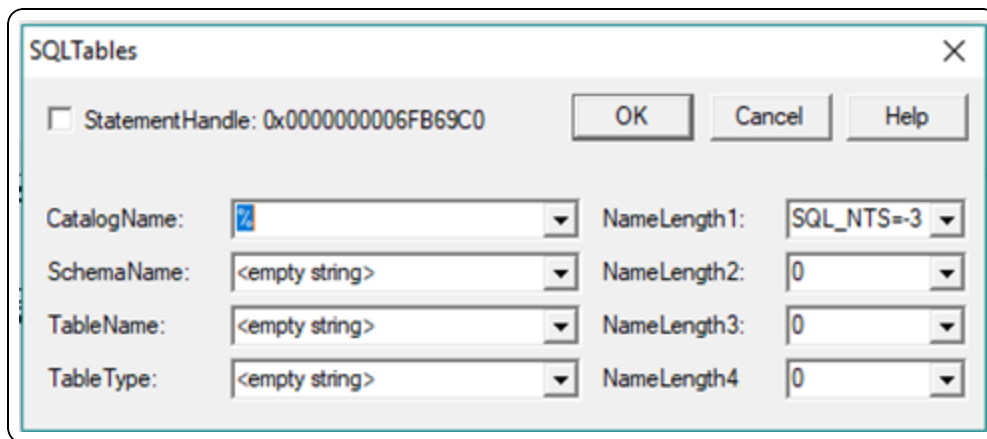
Most ODBC-enabled applications require a connector to implement the following catalog functions. You may wish to implement additional catalog functions in your custom connector.


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](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 to return the catalog metadata about your data store to the ODBC application for the ODBC catalog functions.

Open the file `ULDataEngine.cpp` and navigate to the line **TODO #8 Create and return your Metadata Sources**.

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](#) 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
<code>DSI_TABLES_METADATA</code>	List of all tables defined in the data source.
<code>DSI_CATALOGONLY_METADATA</code>	List of all catalogs defined in the data source, if catalogs are supported.
<code>DSI_SCHEMAONLY_METADATA</code>	List of all schemas defined in the data source. This source is constructed via the <code>ULMetadataHelper</code> 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.

i Tip:

On Linux, Unix, and macOS platforms, this metadata is available using the `datatypes` command in the `iodbctest` utility. As well, the `SQLTables` catalog function is available using the `tables` command.

You can use these commands to test your implementation of Day Three.

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

Open the file `ULDataEngine.cpp` and navigate to the line **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

Open the file `ULQueryExecutor.cpp` and navigate to the line **TODO #10: Implement a QueryExecutor** to go 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

Open the file `ULQueryExecutor.cpp` and navigate to the line **TODO #11: Provide parameter information** to go 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

Open the file `ULQueryExecutor.cpp` and navigate to the line **TODO #12: Implement Query Execution** to go 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

Open the file `ULResultSet.cpp` and navigate to the line **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](#).

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. Open the file `ULDriver.cpp` and navigate to the **TODO #14 Register the `ULMessages.xml` file for handling by `DSIMessageSource` message** to go 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. Open the file `ULDriver.cpp` and navigate to the **TODO #15 Set the vendor name, which will be prepended to error messages** message to jump 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 they create a new ODBC DSN or configure an existing one. Note that the UltraLight connector project for Linux and UNIX platforms does not contain an example ODBC configuration dialog.

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.

Driver Managers

Unlike Windows machines, most Linux, Unix, and macOS installations do not come with a driver manager as part of the operating system. You must install your own driver manager before you can compile and test your connector under Linux, Unix, or macOS. The following driver managers are supported by the Simba SDK:

Driver Manager	Download Location	Simba SDK Support
iODBC	www.iodbc.org	<ul style="list-style-type: none">• Linux and Unix SDK• MacOS SDK
UnixODBC	www.unixodbc.org	<ul style="list-style-type: none">• Linux and Unix SDK

Note:

This document uses the iODBC driver manager as an example, because it is supported by the Simba SDK on all Linux, Unix, and macOS platforms. It also contains an ODBC test utility.

How do I know where my driver manager is installed?

If you did not install the driver manager yourself, you can look for it in typical installation directories. The driver manager must be installed to a directory that is on the library path:

- LD_LIBRARY_PATH on most Linux platforms
- SHLIB_PATH on HP/UX
- LIBPATH on AIX
- DYLD_LIBRARY_PATH on macOS

The iODBC driver manager is often installed to `/usr/lib` or `/usr/local/lib`. If you do not know where your driver manager is installed, try searching those directories for libraries containing the name `libiodb`.

Locating the Configuration Files

The driver manager loads configuration information from the first `odbc.ini` and `odbcinst.ini` files that it finds. The Simba SDK loads connector-specific configuration information from the first `.simba.UltraLight.ini` file that it files. Having multiple copies of these configuration files is not uncommon on development or customer machines, but can lead to confusion when a connector is not loaded as expected.

The Simba SDK installer configures the `odbc.ini` and `odbcinst.ini` files with information for the sample connectors, making it easy for you to get started with your custom connector. However, there are many different locations where the configuration files can be stored, and it is possible for multiple versions of these files to exist on a customer's machine.

Locating the `odbc.ini` File

The driver manager looks for this file in the following locations, in the order specified:

1. If the **ODBCINI** environment variable is set, the driver manager looks in the directory specified by this variable.
2. If there is no variable set, or if no file is found in that location, the driver manager looks in the user's home directory, `~/`.

Note:

In this directory the file must have a preceding dot, for example:
`~/odbc.ini`

3. The driver manager looks in the user-specific Library directory, `~/Library/ODBC`.
4. The driver manager looks in the `/etc` directory, for example `/etc/odbc.ini`.

Note:

There is no dot in front of the `odbc.ini` file in the `/etc` directory.

5. The driver manager looks in the system-wide Library directory, `/Library/ODBC`.

Locating the `odbcinst.ini` File

The driver manager looks for this file in the following locations, in the order specified:

1. If the corresponding environment variable is set, the driver manager looks in the directory specified by this variable.
2. If there is no variable set, or if no file is found in that location, the driver manager looks in the user's home directory.

Note:

In this directory the file must have a preceding dot, for example:
`~/odbcinst.ini`

3. The driver manager looks in the user-specific Library directory,
`~/Library/ODBC.`
4. The driver manager looks in the `/etc` directory, for example
`/etc/odbcinst.ini`

Note:

There is no dot in front of the `odbcinst.ini` file in the `/etc` directory.

5. The driver manager looks in the system-wide Library directory,
`/Library/ODBC.`

Locating the `.simba.UltraLight.ini` configuration file

The Simba SDK searches for the `.simba.UltraLight.ini` file in the following locations, in the specified order:

1. The path, including the file name, specified using the **SIMBAINI** environment variable.

Note:

The name of this environment variable can be rebranded.

2. The connector directory, as a non-hidden `.ini` file.
3. The directory that the client application is launched from.
4. In `$HOME`, as a hidden `.ini` file
5. In `/etc/` as a non-hidden `.ini` file

Using Environment Variables to Specify the Location of the Configuration Files

The location of configuration files is determined by environment variables, as shown below:

Environment Variable	File
ODBCINI	Specifies the full path (including the file name) of the <code>odbc.ini</code> file for the iODBC and unixODBC driver managers.
ODBCINSTINI	Specifies the full path (including the file name) of the <code>odbcinst.ini</code> file for the iODBC driver manager.

For example, these environment variables could be set as shown below:

Example:

```
export ODBCINI=/usr/local/odbc/myodbc.ini
export ODBCINSTINI=/usr/local/odbc/myodbcinst.ini
```

If the environment variables are not set, the driver manager assumes that the configuration files exist in the user's home directory using the default file names `.odbc.ini` and `.odbcinst.ini`.

[Configure the Connector and Data Source](#)

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. you are running.

Install the Evaluation License on Unix, Linux, and macOS

To license the Simba SDK:

- Save the license file under the `$HOME` directory, either as a hidden or a non-hidden file. For example:

```
/home/<user_id>/SimbaEngineSDK.lic for a non-hidden file
```

```
/home/<user_id>/.SimbaEngineSDK.lic for a hidden file
```

Troubleshooting

This section contains solutions to common problems.

Specified Driver Could Not Be Loaded

On Linux, Unix, and macOS platforms, the error `Specified Driver Could Not Be Loaded` may be returned when you try to connect to a data source. Here are some of the reasons the driver manager cannot load the specified connector:

- You are using a 64-bit driver manager, but have 32-bit paths defined in the `odbc.ini` or `odbsinst.ini` files
- You define the path to the debug version of the connector, but you built the non-debug version

Unable to Connect to the Connector

In the `odbc.ini` and `odbcinst.ini` files, be sure there are no characters such as spaces or tabs at the end of a value. This can cause failure to connect.

Is My Driver Manager 32-or-64 bit?

You can use the `file` command.

Example:

This version of `iodbctest` (and therefore the iODBC driver manager) is 64-bit:

```
/usr/local/bin> file iodbctest
iodbctest: ELF 64-bit LSB executable, x86-64, version .....
```

Testing and Troubleshooting in the guide [Developing Connectors for SQL-capable Data Stores](#)

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.