insightsoftware

SimbaEngine SDK

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

Version 10.4

July 2026

Copyright

Contact Us

insightsoftware

www.insightsoftware.com

Contents

About This Guide

Purpose

This guide provides comprehensive, step-by-step instructions for creating a custom ODBC driver using the SimbaEngine SDK. By completing this five-day tutorial, you will learn how to modify and customize the included UltraLight sample driver to connect to your own SQL-capable data source.

At the end of five days, you will have a fully functional read-only ODBC driver that connects to your data store and can be used with any ODBC-enabled application.

Target Audience

This guide is intended for software developers who need to:

- Create custom ODBC drivers for proprietary or specialized data sources
- Enable ODBC connectivity for applications that do not natively support it
- Integrate data sources with business intelligence and reporting tools
- Provide standardized database access to custom data stores

Prerequisites

Before beginning this tutorial, ensure you have the following:

Windows Requirements

Requirement	Details
Development Environment	Microsoft Visual Studio 2022 or later (Community, Professional, or Enterprise)
Operating System	Windows 11 or Windows Server 2016/2019/2022/2025
Programming Skills	Proficiency in C++ programming (intermediate level or higher)
Knowledge	Basic understanding of ODBC concepts, SQL syntax, and database fundamentals
Software	SimbaEngine SDK Version 10.4 (licensed)
Testing Tools	Microsoft Data Access SDK 2.8 (includes ODBC Test tool)
Disk Space	Minimum 2 GB free disk space for SDK and development files

Linux Requirements

Requirement	Details
Development Environment	GCC 8.5 or later

Requirement	Details
Operating System	Ubuntu 20.04 LTS or later, RHEL 8 or later, or equivalent Linux distribution
Build Tools	GNU Make, CMake (optional)
Driver Manager	unixODBC 2.3.x or iODBC 3.52.x
Disk Space	Minimum 2 GB free disk space for SDK and development files

macOS Requirements

Requirement	Details
Development Environment	Xcode 26, 16, or 15 with Command Line Tools installed
Operating System	macOS Tahoe 26, macOS Sonoma 14.5+, or macOS Ventura 13.5+
Driver Manager	iODBC Driver Manager installed
Disk Space	Minimum 2 GB free disk space for SDK and development files

Common Requirements (All Platforms)

Requirement	Details
Programming Skills	Proficiency in C++ programming (intermediate level or higher)
Knowledge	Basic understanding of ODBC concepts, SQL syntax, and database fundamentals
Software	SimbaEngine SDK Version 10.4 (licensed)

Important: You must have a valid SimbaEngine SDK license before beginning this tutorial. Contact insightsoftware sales at www.insightsoftware.com if you need to obtain a license.

Supported Compilers and Platforms

Windows

SimbaEngine SDK version 10.4 supports the following Xcode versions on macOS:

Visual Studio Version	Release Date	Status
2026	February 2026	Current (Recommended)
2022	November 2021	Supported

macOS – Supported Xcode Versions

SimbaEngine SDK version 10.4 supports the following Xcode versions on macOS:

Xcode Version	Release Date	macOS Requirement	Status
Xcode 26.x	September 2025	macOS Tahoe 26 or later	Current (Recommended)
Xcode 16.x	September 2024	macOS Sonoma 14.5 or later	Supported
Xcode 15.x	September 2023	macOS Ventura 13.5 or later	Supported (Legacy)

To check your installed Xcode version, open Terminal and type: xcodebuild -version

Linux – Supported Compilers

SimbaEngine SDK version 10.4 supports the following compilers on Linux:

Compiler	Minimum Version	Recommended Version	Status
GCC (g++)	8.5	8.5	Recommended

To check your installed GCC version, open a terminal and type: g++ --version

Document Conventions

This guide uses the following typographical and formatting conventions:

Convention	Meaning	Example
Bold text	User interface elements	Click Build > Build Solution
Monospace font	Code, file names, paths, commands	UltraLight_vs202x.sln
[INSTALLDIR]	SimbaEngine SDK installation directory	C:\Simba Technologies
<YourProjectName>	Placeholder for your custom name	<YourProjectName>.vcxproj
Italics	Emphasis or document titles	SimbaEngine Developer Guide

Tutorial Structure

This tutorial is organized into five progressive days:

Day	Focus Area	Key Topics	Est. Time
Day 1	Environment Setup	Installation, building samples, registry, project setup	4-6 hours
Day 2	Connection Management	Driver singleton, properties, logging, connections	4-6 hours
Day 3	Metadata Implementation	Catalog functions, type info, tables, columns	4-6 hours
Day 4	Query Execution	Query preparation, execution, parameters, results	4-6 hours
Day 5	Productization	Error handling, branding, dialogs, finalization	4-6 hours

Introduction

This guide will show you how to create your own, custom ODBC driver using SimbaEngine. It will walk you through the steps to modify and customize the included UltraLight sample driver. At the end of five days, you will have a read-only driver that connects to your data store.

ODBC is one of the most established and widely supported APIs for connecting to and working with databases. At the heart of the technology is the ODBC driver, which connects an application to the database. For more information about ODBC, see http://www.simba.com/odbc.htm. For complete information on the ODBC 3.80 specification, see the MSDN ODBC Programmer's Reference, available from the Microsoft web site at http://msdn.microsoft.com/en-us/library/ms714562(VS.85).aspx.aspx)

About SimbaEngine

SimbaEngine is a complete implementation of the ODBC specification, which provides a standard interface to which any ODBC-enabled application can connect. The libraries of SimbaEngine hide the complexity of error checking, session management, data conversions and other low-level implementation details. They expose a simple API, called the Data Store Interface API or DSI API, which defines the operations needed to access a data store. Full documentation for SimbaEngine is available on the Simba website at SimbaEngine Documents.

You use SimbaEngine to create an executable file that will be accessed by common reporting applications and to access your data store when SimbaEngine executes an SQL statement. This executable file can be a Windows DLL, a Linux or Unix shared object, a stand-alone server, or some other form of executable. You create a custom-designed DSI implementation (DSII) that connects directly to your data source. Then, you create the executable by linking libraries from SimbaEngine with the DSI implementation that you have written. In the process, the project files or make files will link in the appropriate SimbaODBC and SimbaEngine libraries to complete the driver. In the final executable, the components from SimbaEngine take responsibility for meeting the data access standards while your custom DSI implementation takes responsibility for accessing your data store and translating it to the DSI API.

About the UltraLight Sample Driver

The UltraLight driver is a sample DSI implementation of an ODBC driver, written in C++, which reads hard coded data. For demonstration purposes, the data is represented by a hard-coded table object called the Person table, which will always be returned if an executed query contains the word "SELECT". If the query does not contain the word "SELECT" then a row count of 12 rows will be returned.

The UltraLight driver helps to prototype a DSI implementation for SQL-based data store. You can also use it as the foundation for commercial DSI implementation if you are careful to remove the shortcuts and simplifications that it contains. This is a fast and effective way to get a data access solution to customers.

Architecture Overview

Understanding the SimbaEngine architecture is essential for successful driver development. Implementation begins with the creation of a DSIDriver class which is responsible for constructing a DSIEnvironment. This in turn is used to construct a connection object (DSIConnection implementation) which can then be used for constructing statements (DSIStatement implementations).

The following diagram illustrates the core component implementation:

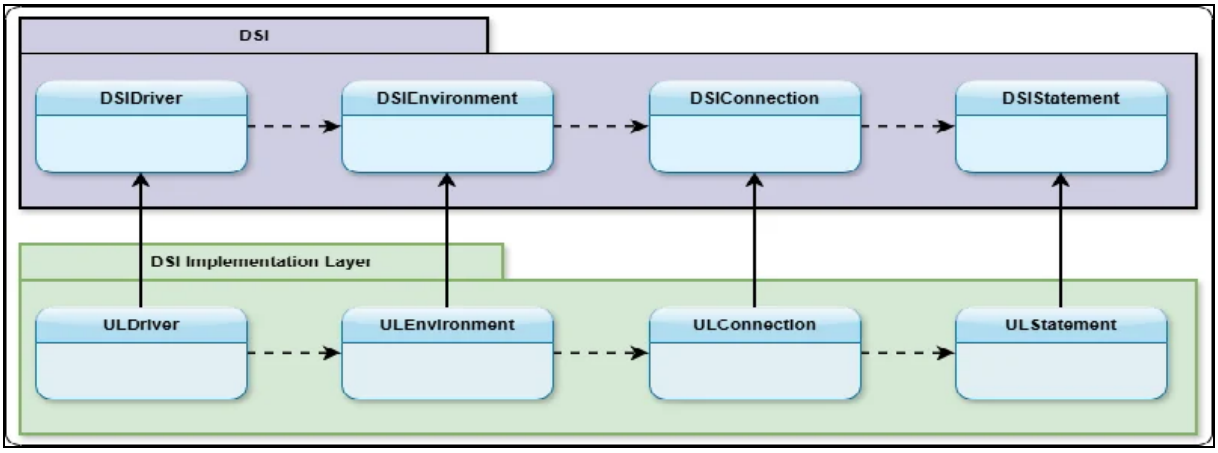

Figure 1: Core Component Implementation

The DSIStatement implementation is responsible for creating a DSIDataEngine object which in turn creates IExecutableOperation objects to execute database operations and return results, and DSIMetadataSource objects to return metadata information.

The following diagram shows the DataEngine implementation:

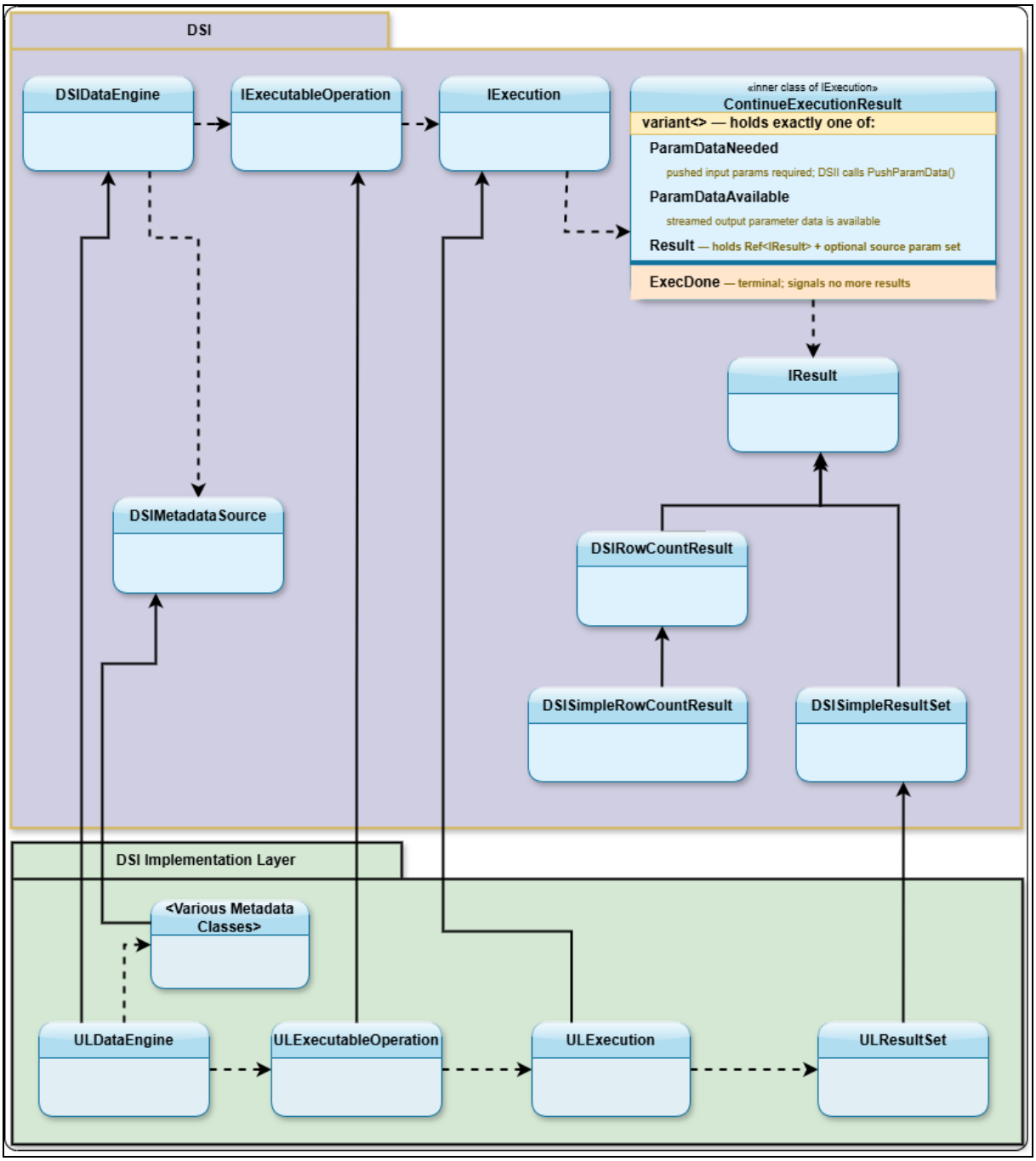

Figure 2: DataEngine Implementation

The final key part of the DSI implementation is to create the framework necessary to retrieve both data and metadata. The IResult class is responsible for retrieving column data and maintaining a cursor across result rows. To implement data retrieval, IResult class interacts directly with 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. The following diagram shows the design pattern for a DSI implementation:

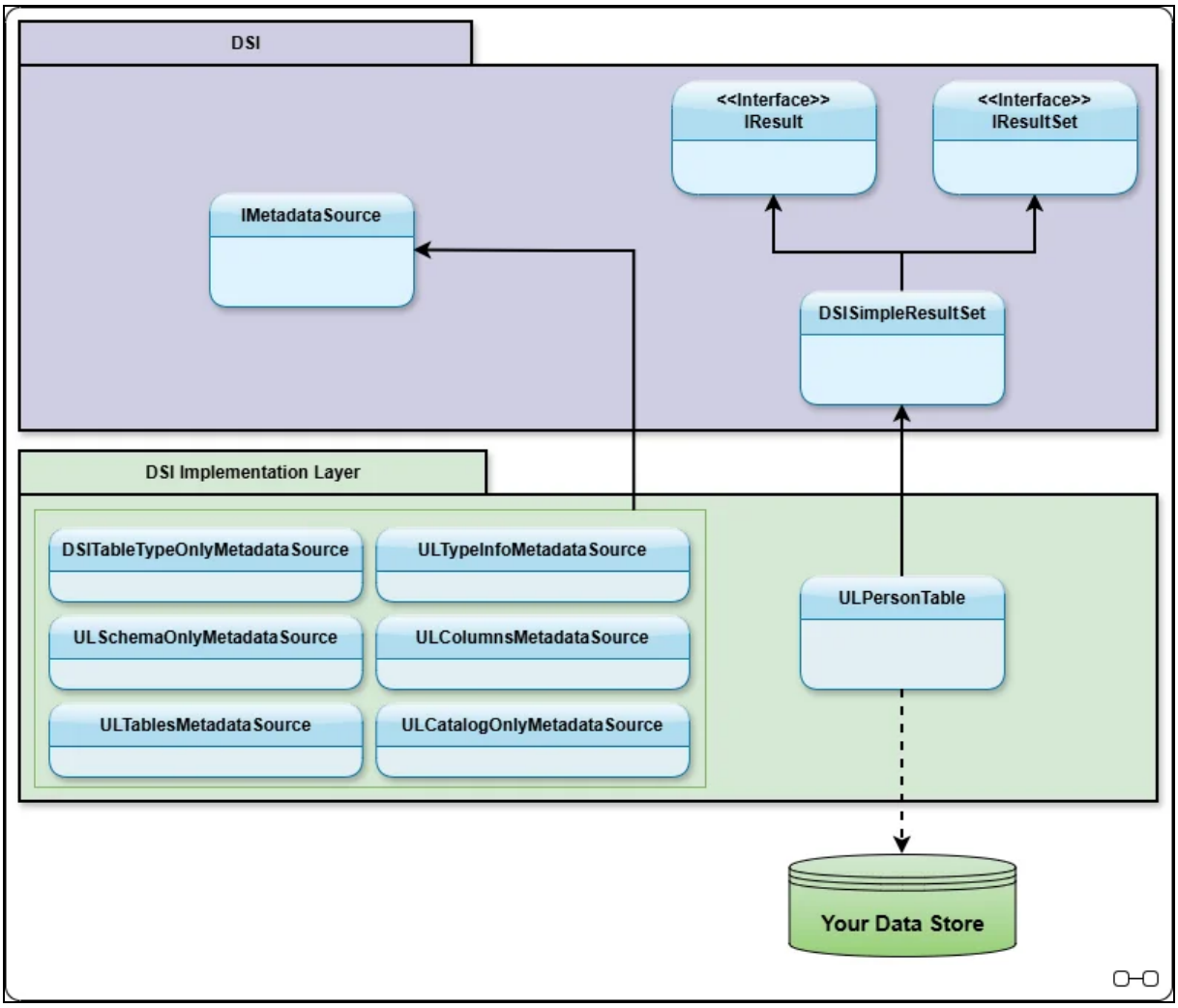

Figure 3: Design Pattern for a DSI Implementation

Development Process Overview

The series of steps to take to get a prototype DSI implementation working with your data store is as follows:

- Set up the development environment
- Make a connection to the data store
- Retrieve metadata
- Work with columns
- Retrieve data

In the UltraLight driver, the areas of the code that you need to change are marked with "TODO" messages along with a short explanatory message. Most of the areas of the code that you need to modify are for productization such as naming the driver, setting the properties that configure the driver,

and naming the XML error file and log files. The other areas of the code that you will modify are related to getting the data and metadata from your data store. Since the UltraLight driver already has the classes and code to do this against its example data store (hard coded data), all you have to do is modify the existing code to make your driver work against your own data store.

Day One: Windows Instructions

Today's task is to set up and test the development environment and project files for the driver. By the end of the day, you will have compiled and tested the first ODBC driver.

LEARNING OBJECTIVES By the end of this section, you will be able to:

- Install and configure the SimbaEngine SDK on Windows
- Build the UltraLight example driver using Visual Studio
- Understand ODBC registry key configuration
- View and configure data sources in ODBC Data Source Administrator
- Test ODBC connectivity using the ODBC Test tool
- Set up a new project for your custom ODBC driver

Install SimbaEngine

- If Visual Studio is running, close it.
- Run the SimbaEngine setup executable that corresponds to your version of Visual Studio and follow the installer's instructions.

Caution: If you have a previous version of SimbaEngine SDK installed, uninstall it before installing the new one. The SimbaEngine environment variables are defined only for the user that ran the installation.

Important: If you install the SDK as a regular user and then run Visual Studio as an administrator, the SDK will not work properly. Ensure you use consistent user permissions throughout the installation and development process.

Platform-Specific Package Names

Platform	Example Package Name
Windows (vs 2022)	SimbaEngineSDK_Release_Windows_vs2022_10.4.*
Windows (vs 2026)	SimbaEngineSDK_Release_Windows_vs2026_10.4.*
Windows (vs 2022)	SimbaEngineSDK_Release_Windows_ARM64_vs2022_10.4.*
Windows (vs 2026)	SimbaEngineSDK_Release_Windows_ARM64_vs2026_10.4.*

Note: ARM is only supported on windows 11.

Build the UltraLight Example Driver

- Launch Microsoft Visual Studio.
- Click **File** > **Open** > **Project** or **Solution**.
- Navigate to `[INSTALLDIR]\SimbaEngineSDK\10.4\Examples\Source\UltraLight\Source` and then open the UltraLight_vs2022.sln file or UltraLight.slnx for vs2026. The default [INSTALLDIR] is `C:\Simba Technologies`.
- Click **Build** > **Configuration Manager** and make sure that the active solution configuration is "Debug_MTDLL" and then click **Close**.
- Click **Build** > **Build Solution** or press **F7** to build the driver.

This will build the debug version of the driver and place it in the following location depending on the Visual Studio version. Path can be bin\windows_vs2022 or bin\windows_vs2026.

For 32-bit drivers:

```
[INSTALLDIR]\SimbaEngineSDK\10.4\Examples\Source\UltraLight\Bin\Windows_
2022\debug64md
```

For 64-bit drivers:

```
[INSTALLDIR]\SimbaEngineSDK\10.4\Examples\Source\UltraLight\Bin\Windows_
vs2022\debug64md
```

Examine the Registry Keys Added by the SimbaEngine Installer

The SimbaEngine installer automatically added or updated the following registry keys that define Data Source Names (DSNs) and driver locations:

Registry Key	Purpose
ODBC Data Sources	Lists each DSN or driver pair.
UltraLightDSII	Defines the Data Source Name (DSN). Used by the ODBC Driver Manager to connect your driver to your database.
ODBC Drivers	Lists the drivers that are installed.
UltraLightDSIIDriver	Defines the driver and its setup location. The ODBC Driver Manager uses this key to connect to and configure your driver.

To view the registry keys:

- Run regedit.exe.
- To view the registry keys that are related to Data Source Names, expand the folders in the Registry Editor to the following location:

  For 32-bit drivers on 32-bit Windows and 64-bit drivers on 64-bit Windows:

  ```
  HKEY_LOCAL_MACHINE/SOFTWARE/ODBC/ODBC.INI
  ```

  For 32-bit drivers on 64-bit Windows:

  ```
  HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ODBC.INI
  ```

- To view the registry keys that are related to ODBC drivers, expand the folders in the Registry Editor to the following location:

  For 32-bit drivers on 32-bit Windows and 64-bit drivers on 64-bit Windows:

  ```
  HKEY_LOCAL_MACHINE/SOFTWARE/ODBC/ODBCINST.INI
  ```

  For 32-bit drivers on 64-bit Windows:

  ```
  HKEY_LOCAL_MACHINE/SOFTWARE/WOW6432NODE/ODBC/ODBCINST.INI
  ```

  Your custom driver installer will have to create similar registry keys.

  Note: Registry keys for 32-bit and 64-bit ODBC drivers are installed in different areas of the Windows registry. See *Appendix B: Windows Registry 32-Bit vs. 64-Bit* for more information.

View the Data Source in the ODBC Data Source Administrator

- Run the Windows ODBC Data Source Administrator.

  Important: ! For 32-bit drivers on 64-bit Windows, you must use the 32-bit ODBC Data Source Administrator.

- In the **ODBC Data Source Administrator**, click the **System DSN** tab.
- Scroll through the list of System Data Sources, select **UltraLightDSII** and then click **Configure**. The Data Source Configuration window opens and displays the fields for the user ID, password, and language.

- Now that you have looked at the configuration information for the driver, click **Cancel** to close the Data Source Configuration window.

Test the Data Source

To test the data source that we have created, you can use any ODBC application, such as, for example, Microsoft Excel, Microsoft Access or ODBCTest. In this section, we will use the ODBC Test tool, which is available in the Microsoft Data Access (MDAC) 2.8 Software Development Kit (SDK).

To download the SDK, visit the following Microsoft Web site:
http://www.microsoft.com/downloads/details.aspx?FamilyID=5067faf8-0db4-429a-b502-de4329c8c850&displaylang=en

- Start the ODBC Test tool. By default, the ODBC Test application is installed in the following folder: `C:\Program Files (x86)\Microsoft Data Access SDK 2.8\Tools\`.
- Navigate to the folder that corresponds to your machine's architecture (arm64 for 64-bit or x86 for 32-bit) and then click `odbcte32.exe` to launch the ANSI version or click `odbct32w.exe` to launch the Unicode version.
- In the ODBC Test tool, select **Conn** > **Full Connect**. The Full Connect window opens.
- Select your Data Source from the list of data sources and then click **OK**. If you do not see your 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 that you created. In other words, if you created a 32-bit data source then you should be using the 32-bit version of the ODBC Test tool.
- When the tool connects to the data source, you will see the message, "**Successfully connected to DSN 'UltraLightDSII'**".

Tip: ✓ It is important to run the correct version of the ODBC Test tool for ANSI or Unicode and 32-bit or 64-bit. Mismatched versions will not show your data source in the connection list. Also ensure your registry is updated correctly.

Set Up a New Project to Build Your Own ODBC Driver

Now that you have built the example driver, you are ready to set up a development project to build your own ODBC driver.

Important: It is very important that you create your own project directory. You might be tempted to just modify the sample project files but we strongly recommend against this, because when you install a new release of the SDK, changes you made will be lost and there may be times, for debugging purposes, that you will need to see if the same error occurs using the sample drivers. If you have modified the sample drivers, this won't be possible without significant additional effort.

- In your Windows Explorer window, copy the `[INSTALLDIR]\SimbaEngineSDK\10.4\Examples\Source\UltraLight` directory and paste it to the same location. This will create a new directory called `UltraLight - Copy`. Rename the directory to something that is meaningful to you. This will be the top-level directory for your new project and DSI implementation files. For the rest of this tutorial, when you see `<YourProjectName>` in the instructions, replace this with the name you choose for this directory which is also the name of your project.

- Open your new directory, open the Source directory and rename the `UltraLight_vs2022.vcxproj or UltraLight.vcxproj` file depending on Visual Studio version in it to `<YourProjectName>.vcxproj` file where you replace `<YourProjectName>` with the name of your project. This will be the project file for your new ODBC driver.

- Rename the `.sln or .slnx` file depending on Visual Studio version. This new `<YourProjectName>.sln` file is the solution file for your new ODBC driver.

- Using a text editor, open the project file (`.vcxproj`) and replace every instance of `UltraLightDSII` in the source code with the name of your new ODBC driver. Then save and close the file.

- Using a text editor, open the solution file (`.sln or slnx`) file depending on the version of visual studio and replace every instance of `UltraLightDSII` in the source code with the name of your new ODBC driver. In addition, references to the name of the project file must be updated to match the `<YourProjectName>.vcxproj` project file that you renamed. Then, save and close the file.

Build Your New Driver

- Launch Microsoft Visual Studio.

- Click **File** > **Open** > **Project** or **Solution**.

- Navigate to `[INSTALLDIR]\SimbaEngineSDK\10.4\Examples\Source\<YourProjectName>\Source` and then open the `<YourProjectName>.sln or .slnx` depending on visual studio version file.

- Click **Build** > **Configuration Manager** and make sure that the active solution configuration is "Debug_MTDLL" and click **Close**.

- Click **Build** > **Build Solution** or press **F7** to build the driver. This will build the debug version of the driver and place it in the location:

  `[INSTALLDIR]\SimbaEngineSDK\10.4\Examples\Source\<YourProjectName>\Bin\windows_vs202x\debug32md`.

- When you build your new project, "TODO" messages appear in the Output window along with the build information. If the Output window is not displayed automatically, you can open it by selecting **Debug** > **Windows** > **Output**.

The following TODO items will appear during the build process. Over the next four days, you will be visiting each "TODO" and modifying the source code:

TODO #	Description	Day
1	Construct driver singleton	2
2	Set the driver properties	2
3	Set the driver-wide logging details	2
4	Set the connection-wide logging details	2
5	Check Connection Settings	2
6	Customize DriverPrompt Dialog	2
7	Establish A Connection	2
8	Create and return your Metadata Sources	3
9	Prepare a Query	4
10	Implement an IExecutableOperation	4
11	Provide parameter information	4
12	Implement IExecution	4
13	Implement your result set	4
14	Register the ULMessages.xml file for handling by DSIMessageSource	5
15	Set the vendor name, which will be prepended to error messages	5

Update the Registry

To update the registry keys, do the following:

1. In Microsoft Visual Studio, click **File** > **Open** > **File** and navigate to `[INSTALLDIR]\SimbaEngineSDK\10.4\Examples\Source\MyUltraLight\Source`.
2. For 32-bit Windows, open **SetupMyUltraLightDSII-32on32.reg**.
   a. For a 32-bit ODBC driver on 64-bit Windows, open **SetupMyUltraLightDSII-32on64.reg**.
   b. For a 64-bit ODBC driver on 64-bit Windows, open **SetupMyUltraLightDSII-64on64.reg**.
3. In the file, replace `[INSTALLDIR]` with the path to the installation directory. In the path, you must enter double backslashes. For example, by default, the samples are installed to "`C:\Simba Technologies`" so in that case, you would replace all instances of `[INSTALLDIR]` with `C:\\Simba Technologies`.
4. Next, update the ODBC Data Sources section to add your new data source. Under the **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\ODBC Data Sources]** section, change `MyUltraLightDSII"="MyUltraLightDSIIDriver` to the name of your new data source and new driver. For example, `<YourProjectName>DSII"="<YourProjectName>DSIIDriver`.
5. Then, modify the data source definition for that data source. Change the line that says **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\MyUltraLightDSII]** so that it contains your new data source name. For example, **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\<YourProjectName>DSII].**
6. Besides the line that starts with "`Driver`"= change the driver name to that of your new ODBC driver. For example, `"Driver"="<YourProjectName>DSIIDriver"`.
7. Update the ODBC Drivers section to add your new driver. Under the **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers]** section, change

"MyUltraLightDSIIDriver"="Installed" to match the name of your new driver. For example, "<YourProjectName>DSIIDriver"="Installed".

8. Modify the driver definition for that driver. Change the line that says **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\MyUltraLightDSIIDriver]** so that it contains your new driver's name. For example, **[HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\<YourProjectName>DSIIDriver]**.
9. Beside the lines that start with "Driver" and "Setup", update the path to the dll file for both.
10. Click **Edit** > **Find and Replace** > **Quick Replace**. Then, replace "UltraLight" in the whole file with the name of your new ODBC driver.
11. Click **Save** and close the file.
12. In the Registry Editor (regedit.exe), click **File** > **Import**, navigate to the registry file that you just modified and then click **Open**. A message is displayed that says that the keys and values have been successfully added to the registry.

View Your New Data Source in the ODBC Data Source Administrator

- Run the Windows ODBC Data Source Administrator.
- In the ODBC Data Source Administrator, click the **System DSN** tab.
- Scroll through the list of System Data Sources, select **<YourProjectName>DSII** and then click **Configure**.

  The **Data Source Configuration** window opens and displays the data source name, description and the data directory.
- Now that you have looked at the configuration information for your new driver, click **Cancel** to close the **Data Source Configuration** window.

Test Your New Data Source

1. Start the ODBC Test tool. By default, the ODBC Test application is installed in the following folder: C:\Program Files (x86)\Microsoft Data Access SDK 2.8\Tools\
2. Attach Visual Studio to the ODBC Test process. To do this, go to **Microsoft Visual Studio** and click **Debug** > **Attach to Process**.
3. In the **Attach to Process** window, select the **ODBC Test process** and click **Attach**. The process name will be either odbc32.exe or odbct32w.exe.
4. Add a breakpoint in Main_Windows.cpp, on the function Simba::DSI::DSIDriverFactory(). This function runs as soon as the Driver Manager loads the ODBC driver.
5. In the ODBC Test tool, select **Conn** > **Full Connect**. The Full Connect window opens.

6. Select your Data Source from the list of data sources and click **OK**. If you do not see your 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 that you created.
7. You should hit the breakpoint you created and focus should switch to Visual Studio.
8. To continue running the program, select **Debug** > **Continue**. The focus returns to the ODBC Test window.
9. Enter `Select * from ULResultSet` in ODBC Test and click the button on the toolbar with the exclamation icon. Then click the button beside it. This will output a simple result set.

Tip: ✓ Day One Checkpoint – Verify Your Progress

- SimbaEngine SDK is installed successfully
- UltraLight example driver builds without errors
- Registry keys are visible in Registry Editor
- Data source appears in ODBC Data Source Administrator
- Successfully connected to UltraLightDSII using ODBC Test tool
- New project directory is created and files renamed
- New driver project builds without errors
- New driver's registry keys are imported successfully
- New driver connects successfully via ODBC Test tool
- Can execute SELECT query and see results in ODBC Test

Summary – Day One

At this point, you have completed the following tasks:

- Install SimbaEngine and build the sample driver included with the SDK.
- Learn about the Windows ODBC Data Source Administrator, the creation of new Data Source names and the area of the Windows Registry where these settings are stored.
- Test the sample drivers using an ODBC-enabled application.
- Set up a new project directory where you will begin to modify one the sample drivers as the starting point for your new driver.

Day One: Linux and macOS Instructions

Today's task is to set up and test the development environment and project files for the driver. By the end of the day, you will have compiled and tested the first ODBC driver.

LEARNING OBJECTIVES: By the end of this section, you will be able to:

- Install and configure the SimbaEngine SDK on Linux and macOS
- Build the UltraLight example driver using makefiles
- Understand ODBC Driver Manager configuration file structure
- Configure data sources in odbc.ini and drivers in odbcinst.ini
- Test ODBC connectivity using isql (unixODBC) or iodbctest (iODBC)
- Set up a new project for your custom ODBC driver
- Debug your driver using GDB (Linux) or LLDB (macOS)

Install SimbaEngine

On Linux and macOS, SimbaEngine is provided as a tar.gz file – a tar format archive that has been compressed using the gzip tool. The file name includes characters that indicate the build number and platform.

Platform-Specific Package Names

Platform	Example Package Name
Linux Arm (64 Bit)	SimbaEngineSDK_Release_Linux-aarch64_gcc8_5_10.4.0.1001.tar.gz
Linux (64 bit)	SimbaEngineSDK_Release_Linux-x86_gcc8_5_10.4.0.1001.tar.gz
macOS (Universal)	SimbaEngineSDK_Release_Darwin-universal_xcode15_2_10.4.0.1001.tar.gz

Caution: If you have a previous version of SimbaEngine installed, uninstall it before installing the new one. The SimbaEngine environment variables are defined only for the user that ran the installation.

Installation Procedure

- Open a terminal window.

Platform	How to Open Terminal
Linux	Search for "Terminal" in your applications menu, or press Ctrl+Alt+T
macOS	Find Terminal in Applications > Utilities > Terminal, or search with Spotlight (Cmd+Space)

- Navigate to the directory where you want to install SimbaEngine.

Platform	Recommended Directory	Command
Linux	/opt	cd /opt
macOS	/Library	cd /Library

Important: On macOS, we recommend installing SimbaEngine to the /Library directory. Microsoft Excel 2016 and later -will only load drivers from /Library and /Applications.

- Copy the SimbaEngineSDK*.tar.gz file to the installation directory.

```
Linux
sudo cp ~/Downloads/SimbaEngineSDK*.tar.gz /opt/


macOS
sudo cp ~/Downloads/SimbaEngineSDK*.tar.gz /Library/


Linux Arm
SimbaEngineSDK_Release_Linux-aarch64_gcc8_5_10.4.0.1001.tar.gz
```

- Uncompress the gzip file:

```
sudo gunzip SimbaEngineSDK*.tar.gz
```

- Extract the tar archive:

```
sudo tar -xvf SimbaEngineSDK*.tar
```

- Verify the installation by listing the extracted contents:

```
ls -la SimbaEngineSDK/10.4
```

You should see the following directories:

Directory	Description
DataAccessComponents/	Core SimbaEngine libraries and headers
Documentation/	Developer guides, API reference, and sample configuration files
Examples/	Sample driver source code, databases, and build files

Note: Throughout this guide, the directory where you install SimbaEngine is referred to using the placeholder `[INSTALLDIR]`. When working through procedures, replace `INSTALLDIR` with your actual installation path (e.g., /opt on Linux or /Library on macOS).

Build the UltraLight Example Driver

On Linux and macOS, the sample drivers include makefiles instead of Visual Studio solution files. The process to build each of the sample drivers is similar across both platforms.

Note: If you wish to maintain your own build scripts, we ship a Perl script located at ${SIMBAENGINE_DIR}/GetBuildInfo.pl which can be used to get the needed values to add to CPPFLAGS & libraries to link to, see the GetBuildInfo.README file in the same directory for more information.

Output Library Types

Platform	Library Extension	Example Output
Linux	.so (shared object)	libUltraLight_debug.so
macOS	.dylib (dynamic library)	libUltraLight_debug.dylib

Set Environment Variables

- Set the SIMBAENGINE_DIR environment variable:

```
export SIMBAENGINE_DIR=[INSTALLDIR]
/SimbaEngineSDK/10.4/DataAccessComponents
```

  Replace `<INSTALLDIR>` with your actual installation directory (e.g., /opt on Linux or /Library on macOS).

- Set the SIMBAENGINE_THIRDPARTY_DIR environment variable:

```
export SIMBAENGINE_THIRDPARTY_DIR=[INSTALLDIR]
/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty
```

Build the Driver

- Change to the Makefiles directory:

```
cd [INSTALLDIR]
/SimbaEngineSDK/10.4/examples/Source/Ultralight/Source/
```

- Run the makefile for the debug target:

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

Note: On macOS, if you are using a specific Xcode version, you can specify the COMPILER option:

- For Xcode 26: make -f UltraLight.mak COMPILER=Xcode26 debug
- For Xcode 16: make -f UltraLight.mak COMPILER=Xcode16 debug
- For Xcode 15: make -f UltraLight.mak COMPILER=Xcode15 debug

This will build the debug version of the driver and place it in the following location:

Platform	Output Directory
Linux	[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/UltraLight/Bin/Linux_x86__gcc85/
macOS	[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/UltraLight/Bin/Darwin_multiarch_Xcode15/
Linux arm	[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/UltraLight/Bin/Linux_aarch64_gcc85/

Build Targets

Target	Command	Output Library
Debug	mk.sh MODE=debug for debug	libUltraLight_debug.so / .dylib
Release	mk.sh MODE=release for release	libUltraLight.so / .dylib
Debug Server	mk.sh MODE=debug BUILDSERVER=1 for debug	Server executable (debug)
Release Server	mk.sh MODE=release BUILDSERVER=1 for release	Server executable (release)

Understanding Universal Libraries (macOS Only)

On macOS, the build produces a universal (fat) binary that contains code for multiple architectures (x86_64 for Intel and arm64 for Apple Silicon). You can use the lipo command to inspect or extract architecture-specific libraries.

To display architectures in the universal library:

```
cd [INSTALLDIR]
/SimbaEngineSDK/10.4/Examples/Source/UltraLight/Bin/Darwin_multiarch_
Xcode15

lipo -info libUltraLight_debug.dylib
```

The output will show:

```
Architectures in the fat file: libUltraLight_debug.dylib are: x86_64 arm64
```

Configure the ODBC Data Source and Driver

Both Linux and macOS use configuration files to define ODBC data sources and drivers. The configuration process is similar, though the Driver Manager may differ.

Supported Driver Managers

Driver Manager	Configuration Files
unixODBC and iODBC.	odbc.ini, odbcinst.ini

Note: We recommend unixODBC, except on macOS, where the standard is iODBC.

Configuration File Locations

Driver Manager	DSN Configuration	Driver Configuration
unixODBC	/etc/odbc.ini or ~/.odbc.ini	/etc/odbcinst.ini or ~/.odbcinst.ini
iODBC	~/.odbc.ini	~/.odbcinst.ini

Tip: ✓ You can override the default configuration file locations using environment variables:

- ODBCINI – Path to odbc.ini file
- ODBCINSTINI – Path to odbcinst.ini file
- ODBCSYSINI – Directory containing system odbc.ini (unixODBC only)

For sample configuration files, see the following directory:

```
[INSTALLDIR]/SimbaEngineSDK/10.4/Documentation/Setup
```

Configure an ODBC Data Source

The odbc.ini (or .odbc.ini) configuration file defines ODBC Data Sources.

To configure a data source:

1. Check if the configuration file already exists

```
Linux (unixODBC system-wide)
ls -la /etc/odbc.ini

Linux or macOS (user-specific)
ls -la ~/.odbc.ini
```

2. If the file does not exist, copy the sample file:

```
Linux (user-specific)
cp [INSTALLDIR]/SimbaEngineSDK/10.4/Documentation/Setup/odbc.ini
~/.odbc.ini

macOS
cp [INSTALLDIR]/SimbaEngineSDK/10.4/Documentation/Setup/odbc.ini
~/.odbc.ini
```

3. Open the configuration file in a text editor and ensure the following entries exist:

```
[ODBC Data Sources]

UltraLightDSII=UltraLightDSIIDriver

[UltraLightDSII]

Description=Sample SimbaEngine UltraLight DSII

Driver=
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/UltraLight/Bin/<platform
>/libUltraLight.so

Locale=en-US
```

Important: Replace [INSTALLDIR] with your actual installation path and <platform> with:

- Linux_x86_64_gcc for Linux
- Darwin_universal_clang for macOS Also change .so to .dylib for macOS.

Define an ODBC Driver

ODBC Drivers are defined in the odbcinst.ini (or .odbcinst.ini) configuration file.

To define a driver:

- Check if the configuration file exists and open it (or create it):

```
Linux (user-specific) or macOS
ls -la ~/.odbcinst.ini
```

- If the file does not exist, copy the sample file:

```
cp [INSTALLDIR]/SimbaEngineSDK/10.4/Documentation/Setup/odbcinst.ini
~/.odbcinst.ini
```

- Ensure the following entries exist:

```
[ODBC Drivers]

UltraLightDSIIDriver=Installed

[UltraLightDSIIDriver]

APILevel=1

ConnectFunctions=YYY

Description=Sample SimbaEngine UltraLight DSII

Driver=
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/UltraLight/Bin/<plat
form>/libUltraLight.so

DriverODBCVer=03.80

SQLLevel=1
```

Note: See https://learn.microsoft.com/en-us/sql/odbc/reference/install/driver-specification-subkeys?view=sql-server-ver17 for more information about 'DriverODBCVer' & 'SQLLevel'

Configure the Driver Settings (.ini file)

Configuration options for drivers created using SimbaEngine appear in a separate .ini file. SimbaEngine searches for this file in the following order:

1. The path specified by the SIMBAINI environment variable
2. The driver directory, as a non-hidden .ini file
3. The user's home directory, as a hidden .ini file (e.g. .simba.ultralight.ini)
4. The /etc/ directory as a non-hidden .ini file

To configure the driver settings:

- Copy the sample configuration file to your home directory:

```
cp
[INSTALLDIR]/SimbaEngineSDK/10.4/Documentation/Setup/.simba.ultraligh
t.ini ~/
```

- Open the file and configure the following settings:

Setting	Description	Example Value(s)
DriverManagerEncoding	Character encoding used by the Driver Manager	Only the following values are accepted (case-sensitive): ■ UTF-8 ■ UTF-16 ■ UTF-32 If not specified, or set to an empty value, the SDK will fall back to attempting to infer the encoding based on its built-in Driver-Manager-Detection logic, which assumes the driver manager in use was built with default settings.
ErrorMessagesPath	Path to the error messages XML file	[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/UltraLight/ErrorMessages
ODBCInstLib	Path to the ODBC installer library. If not given an absolute path, resolution depends on LD_LIBRARY_PATH/DYLD_LIBRARY_PATH.	■ libiodbcinst.so ■ libiodbcinst.dylib ■ libodbcinst.so ■ /etc/blah/dm/unixodbc-3.23/libodbcinst.so

For more information about driver configuration and which encoding to use for which driver manager and platform, refer to the SimbaEngine Developer Guide.

Test the Data Source

Before testing, ensure that the required libraries are in your library path.

Set the Library Path

Platform	Environment Variable	Command
Linux	LD_LIBRARY_PATH	export LD_LIBRARY_PATH=<path_to_libs>:$LD_LIBRARY_PATH
macOS	DYLD_LIBRARY_PATH	export DYLD_LIBRARY_PATH=<path_to_libs>:$DYLD_LIBRARY_PATH

Add the ICU and OpenSSL library paths:

Linux:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:\
[INSTALLDIR]/SimbaEngineSDK/10.3/DataAccessComponents/ThirdParty/icu/74.2/Linux_x86_gcc85/release64/lib
[INSTALLDIR]/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty/openssl/3.5/Linux_x86_gcc85/release64/lib
```

Linux ARM:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:\
[INSTALLDIR]/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty/icu/78.1/Linux_aarch64_gcc85/release64/lib
[INSTALLDIR]/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty/openssl/3.5/Linux_aarch64_gcc85/release64/lib
```

macOS (Xcode 26):

```
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:\
[INSTALLDIR]/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty/icu/78.1/Darwin_multiarch_Xcode15/release64-universal/lib
[INSTALLDIR]/SimbaEngineSDK/10.4/DataAccessComponents/ThirdParty/openssl/3.5/Darwin_multiarch_Xcode15/release64-universal/lib
```

Note: The exact paths may vary depending on the SDK build and third-party library versions. Check the ThirdParty directory for available paths.

Test Using the Command-Line Tools

Driver Manager	Test Tool	Command
unixODBC	isql	isql -v UltraLightDSII <username> <password>
iODBC	iodbctest	iodbctest

Testing with unixODBC (isql)

1. At the command prompt, type:

```
isql -v UltraLightDSII <YourUserName> <YourPassword>
```

2. If successful, you will see a SQL> prompt. Type a query:

```
SELECT * FROM ULResultSet
```

Testing with iODBC (iodbctest)

- At the command prompt, type:

```
iodbctest
```

- At the "Enter ODBC connect string" prompt, type ? to list available DSNs, then connect:

```
DSN=UltraLightDSII;UID=<YourUserName>;PWD=<YourPassword>
```

- If successful, you will see a SQL> prompt. Type a query:

```
SELECT * FROM ULResultSet
```

Tip: ✓ If the connection is successful, you are ready to start building your own ODBC driver.

Set Up a New Project for Your ODBC Driver

Important: It is very important that you create your own project directory. Do not modify the sample project files directly, because:

- Changes will be lost when you install a new SDK release.
- You may need the original sample drivers for debugging and comparison.

1. Copy the UltraLight directory to create your new project:

```
cp -R [INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/UltraLight \
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/MyUltraLight
```

2. Build your new driver:

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/MyUltraLight/Makefiles
mk.sh MODE=debug
```

Configure Your New Driver

- Add your new data source to the odbc.ini file:

```
[ODBC Data Sources]
MyUltraLightDSII=MyUltraLightDSIIDriver
[MyUltraLightDSII]
Description=My Custom UltraLight Driver
Driver=
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/MyUltraLight/Bin/<pl
atform>/libMyUltraLight_debug.so
Locale=en-US
```

- Add your new driver to the odbcinst.ini file:

```
[ODBC Drivers]
MyUltraLightDSIIDriver=Installed
[MyUltraLightDSIIDriver]
Driver=
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/MyUltraLight/Bin/<pl
atform>/libMyUltraLight_debug.so
```

- Copy and rename the driver configuration file:

```
cp ~/.simba.ultralight.ini ~/.simba.myultralight.ini
```

- Test your new driver using isql or iodbctest.

Debugging Your Driver

You can use a debugger to step through your driver code while testing.

Platform-Specific Debuggers

Platform	Debugger	Command to Start
Linux	GDB (GNU Debugger)	gdb isql or gdb iodbctest
macOS	LLDB	lldb iodbctest

Debugging with GDB (Linux)

- Start GDB with the test tool:

gdb isql

- Set a breakpoint at the driver entry point:

```
(gdb) break Simba::DSI::DSIDriverFactory
```

- Run the program with connection arguments:

```
(gdb) run -v MyUltraLightDSII username password
```

Debugging with LLDB (macOS)

- Start LLDB with the test tool:

```
lldb iodbctest
```

- Set a breakpoint at the driver entry point:

```
(lldb) b Simba::DSI::DSIDriverFactory
```

- Run with a connection string:

```
(lldb) run DSN=MyUltraLightDSII;UID=<username>;PWD=<password>
```

> **Note:** The driver is not loaded until you make a connection. Breakpoints may show as "pending" until the driver library is loaded by the Driver Manager.

Common Debugger Commands

Action	GDB Command	LLDB Command
Set breakpoint	break <function>	b <function>
Run program	run [args]	run [args]
Continue execution	continue	c
Step over	next	n
Step into	step	s
Step out	finish	finish
Print variable	print <var>	p <var>
Backtrace	backtrace	bt
Quit	quit	quit

✓ **Day One Checkpoint – Verify Your Progress**:

- SimbaEngine SDK is installed successfully
- Environment variables are set correctly
- UltraLight example driver builds without errors
- Configuration files (odbc.ini, odbcinst.ini) are properly configured
- Driver settings file (.simba.ultralight.ini) is configured

- Successfully connected to UltraLightDSII using isql or iodbctest
- New project directory is created and files renamed
- New driver project builds without errors
- New driver's configuration files are updated
- New driver connects successfully
- Can execute SELECT query and see results

TODO Items Reference

When you build your new project, "TODO" messages appear in the build output. Over the next four days, you will visit each TODO and modify the source code:

TODO #	Description	Day
1	Construct driver singleton	2
2	Set the driver properties	2
3	Set the driver-wide logging details	2
4	Set the connection-wide logging details	2
5	Check Connection Settings	2
6	Customize DriverPrompt Dialog	2
7	Establish A Connection	2
8	Create and return your Metadata Sources	3
9	Prepare a Query	4
10	Implement an IExecutableOperation	4
11	Provide parameter information	4
12	Implement IExecution	4
13	Implement your result set	4
14	Register the Messages.xml file for handling by DSIMessageSource	5
15	Set the vendor name, which will be prepended to error messages	5

Summary – Day One

At this point, you have completed the following tasks:

- Install SimbaEngine SDK on Linux or macOS and build the sample driver
- Learn about ODBC Driver Manager configuration files (odbc.ini, odbcinst.ini) and how Data Source Names and drivers are configured
- Test the sample drivers using isql (unixODBC) or iodbctest (iODBC)

- Set up a new project directory as the starting point for your custom driver
- Debug your driver using GDB (Linux) or LLDB (macOS)

Day Two: Connection Management

Today's goal is to customize the connector, enable logging and establish a connection to the data store. To accomplish this, check TODO items 1 to 7.

LEARNING OBJECTIVES: By the end of this section, you will be able to:

- Understand and modify the Simba::DSI::DSIDriverFactory() entry point
- Configure driver properties and character encodings
- Set up driver-wide and connection-wide logging
- Validate and process connection settings
- Customize the driver prompt dialog for user interaction
- Establish authenticated connections to your data store

Remember that, when the project is built, you will see the TODO messages in the Output window. To rebuild the whole solution, select **Build** > **Rebuild Solution**. If it does not display, open the Output window by selecting **Debug** > **Windows** > **Output**. Double click the TODO relevant section of code.

Construct a Connector Singleton

TODO #1: Construct a Connector singleton

The `Simba::DSI::DSIDriverFactory()` implementation in MainWindow.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.

Windows

1. Launch Microsoft Visual Studio.
2. Click **File** > **Open** > **Project/Solution**.
3. Navigate to [INSTALLDIR]\SimbaEngineSDK\10.4\Examples\Source\<YourProjectName>\Source and then open the <YourProjectName>_VS2022.vcxproj or <YourProject>.vcxproj file.
4. Rebuild your solution and double-click the TODO #1 message in the Output window. The Main_Windows.cpp file opens.
5. Look at the Simba::DSI::DSIDriverFactory() implementation.
6. Specify the location that is used when reading driver settings from the registry. This change is related to rebranding. Locate the line of code where the #define directive specifies DRIVER_WINDOWS_BRANDING and replace the value with something like "Company\\Driver" where "Company" is your company name and "Driver" is the name of your driver.
7. Add any initialization logic needed for your connector before constructing your Simba::DSI::IDriver implementation.
8. Click **Save**.

Linux and macOS

1. Using your preferred text editor or IDE, open the Main_Unix.cpp file located in:

   ```
   [INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/
   ```

2. Search for TODO #1 or locate the Simba::DSI::DSIDriverFactory() implementation.
3. Look at the Simba::DSI::DSIDriverFactory() implementation.
4. Specify the branding string that is used when reading driver settings from the .ini file. This change is related to rebranding. Locate the line of code where the #define directive specifies DRIVER_LINUX_BRANDING (Linux) or DRIVER_OSX_BRANDING (macOS) and replace the value with something like "company.driver" where "company" is your company name and "driver" is the name of your driver. For example:
   - Change #define DRIVER_LINUX_BRANDING "simba.ultralight" to #define DRIVER_LINUX_BRANDING "mycompany.mydriver"
5. Add the processing, if you are building a commercial connector.
6. Save the file.
7. Rebuild your driver by running the makefile:

   ```
   cd [INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/<YourProjectName>/Source
                           mk.sh MODE=DEBUG
   ```

Understanding Driver Branding

Platform	Branding Macro	Effect
Windows	DRIVER_WINDOWS_BRANDING	Registry node for driver settings
Linux	DRIVER_LINUX_BRANDING	Name of the .ini file for settings
macOS	DRIVER_LINUX_BRANDING	Name of the .ini file for settings

Note: This step distinguishes your driver from samples and other drivers. On Windows, this changes the registry node (e.g., **HKEY_LOCAL_MACHINE\SOFTWARE\MyCompany\MyDriver**). On Linux/macOS, this changes the .ini file name (e.g., .mycompany.mydriver.ini).

Set the Connector Properties

TODO #2: Set the driver properties

Windows:

1. Double-click the TODO #2 message in the Output window. The ULDriver.cpp file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the ULDriver.cpp file located in:

   ```
   [INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/
   ```

2. Search for TODO #2 or locate the SetDriverPropertyValues() method.

All Platforms:

1. Look at SetDriverPropertyValues() and set up the general properties for the connector.
2. Change the DSI_DRIVER_DRIVER_NAME setting. Set this to the name of your driver (the same name you used to replace on Day One).
3. Depending on the character sets or Unicode encoding used on your data store, you may want to change the following settings (Note these are defaults; per-column encodings are possible if required):

Setting	Description	Default Value
DSI_DRIVER_STRING_DATA_ENCODING	The encoding of character data within the data store	Based on checking driver and system configuration in the following order: ■ SIMBA_APP_ANSI_ENCODING environment variable ■ ANSIENCODING driver configuration setting ■ ICU's ucnv_getDefaultName() function, which typically determines it from the system locale configuration ■ LC_CTYPE or LANG environment variables on many *nix systems ■ Windows default app language settings
DSI_DRIVER_WIDE_STRING_DATA_ENCODING	The encoding of wide character data within the data store	■ Windows/macOS: ENC_UTF16_LE ■ Linux: ENC_UTF32_LE

4. Save the file.

Linux and macOS only:

- Rebuild your driver by running the makefile.

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source
```

```
mk.sh MODE=DEBUG
```

Set the Logging Details

Customize the driver logs errors and other information. The important loggers are the ***driver log*** for anything not specific to a single connection, and the ***connection log*** for anything unique to a single connection or statement within a connection. Following the TODOs below, you can use our provided logger implementation and just rename the output filename. Or you may entirely replace it later with your own implementation of ILogger interface.

TODO #3: Set the driver-wide logging details

TODO #4: Set the connection-wide logging details

Linux and macOS:

- Using your preferred text editor or IDE, open the ULDriver.cpp file (if not already open) and search for TODO #3.

All Platforms:

1. Change the driver log's file name to something meaningful for your driver.
2. Locate TODO #4 (in the same file or search for it).
3. The connections currently use the same log file as the driver. You may choose to have each connection create a separate log file. If so, change the code to create a DSIFileLogger with a unique log file name.
4. Save the file.

Linux and macOS only:

- Rebuild your driver.

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/So
urce

mk.sh MODE=DEBUG
```

Tip: ✓ By default, the SimbaEngine UltraLight Driver maintains a log file for the entire driver. If you require more fine-grained logging, then consider one for all driver-based calls and one for each connection created as noted in step 4, above. For more information about how to enable logging, refer to the SimbaEngine Developer Guide.

Check the Connection Settings

TODO #5: 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 sent to the *ULConnection::UpdateConnectionSettings()* method which is responsible for verifying that all the required, and any optional, connection settings are present. Validating the correctness (e.g., passwords) must be done later in the Connect() method, as new values can be added between the calls, but you can check here as well if desired.

For example, the connection string *DSN=UltraLight;UID=user* will be broken down into key-value pairs and passed through *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.

Windows:

- Double-click the TODO #5 message in the Output window. The ULConnection.cpp file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the ULConnection.cpp file located in:

   `[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`

2. Search for TODO #5 or locate the UpdateConnectionSettings() method.

All Platforms :

1. The *UpdateConnectionSettings*() function should validate that the key-value pairs are sufficient to create a connection, and any settings that are not present should be added to the DSIConnSettingResponseMap parameter.

2. The *VerifyRequiredSetting*() or *VerifyOptionalSetting*() utility functions can be used to perform this verification and will add missing settings to *DSIConnSettingResponseMap*. For example, the UltraLight driver verifies that the entries within in_*connSettingsHelper* are sufficient to create a connection by using the following code:

   The `VerifyRequiredSetting() or VerifyOptionalSetting()` utility functions can be used to perform this verification and will add missing settings to `DSIConnSettingResponseMap`. For example, the UltraLight driver verifies that the entries within in_connectionSettings are sufficient to create a connection, by using the following code:

```
VerifyRequiredSetting(UL_UID_KEY, in_connSettingsHelper.GetSettings(),
out_connectionSettings); VerifyRequiredSetting(UL_PWD_KEY, in_
connSettings Helper.GetSettings(), out_connectionSettings);
VerifyOptionalSetting(UL_LNG_KEY, in_connSettings Helper.GetSettings(),
out_connectionSettings);
```

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

The settings can alternatively be verified manually. If the entries within in_connSettingsHelper are not sufficient to create a connection, then you 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.

3. Save the file.

Linux and macOS:

- Rebuild your driver.

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/So
urce

mk.sh MODE=DEBUG
```

Customize the DriverPrompt Dialog

TODO #6: Customize DriverPrompt Dialog

Depending on how the connection was initiated by the application, the SDK may call *ULConnection::PromptDialog()* to allow the user to specify more information. In general, if there are any required settings present in the *DSIConnSettingResponseMap*, then *PromptDialog*() will be called. Note that, if the application requests, *PromptDialog*() may not be called in this case or may be called even if there are no settings in the *DSIConnSettingResponseMap*. Also, note that the SDK will call Simba::DSI::IConnection::SetAppDialogInfo() and Simba::DSI::IConnection::CanPrompt() before attempting to call PromptDialog(), so you can override these for finer control.

Platform	Dialog Behavior
Windows	ULConnection::PromptDialog() displays a configuration dialog box which is shown by the Windows ODBC Data Source Administrator when configuring the driver.
Linux	The iODBC or unixODBC Administrator, or a custom dialog implementation, may be used for user prompts.
macOS	The iODBC Administrator or a custom dialog implementation may be used for user prompts.

The method takes in the following parameters:

Parameter	Description
in_ connResponseMap	A connection response map which can be populated with settings which have not been entered by the user. This is then used by the driver to notify the user that information is missing. Currently this variable is unused in the sample.
io_ connSettingRequest	An object which is populated by the method with settings entered into the dialog by the user.
in_parentWindow	The handle to the parent window to make the prompt window a child of. On Linux

Parameter	Description
	and macOS, this parameter may be unused or handled differently depending on your dialog implementation.
in_promptType	An enum specifying if only required fields are to be available, or if optional fields should be available as well. For example, in the UltraLight driver, the language ("LNG") is an optional field.

Tip: ✓ This step may be deferred until later to speed up the initial development of your driver. To defer it, leave the TODO in place and modify the method to return false. Until you return and implement this, you will need to ensure that you always provide complete connection information in your connection string or DSN settings.

To customize the connection prompt dialog:

Windows:

- Double-click the TODO #6 message in the Output window. The ULConnection.cpp file opens

  `[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`

Linux and macOS:

1. Using your preferred text editor or IDE, open the ULConnection.cpp file located in:
2. Search for TODO #6 or locate the PromptDialog() method.

All Platforms:

1. The dialog and the related code in this method can be modified to take in different parameters as required by your driver.
2. Save the file.

Linux and macOS only:

- Rebuild your driver.

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source

mk.sh MODE=DEBUG
```

Note: On Windows, the dialog is typically implemented as a Win32 GUI dialog. On Linux and macOS, GUI dialogs require additional frameworks (such as Qt or GTK on Linux, or Cocoa on macOS). If your driver is used primarily with applications that provide their own connection dialogs or connection string builders, you may choose to implement minimal prompt functionality on non-Windows platforms. Consult the SimbaEngine Developer Guide for platform-specific dialog implementation guidance.

Establish a Connection

TODO #7: Establish A Connection

Once *ULConnection::UpdateConnectionSettings*() returns *out_connectionSettings* without any required settings (if there are only optional settings, a connection can still occur), the Simba ODBC layer will call *ULConnection::Connect*() passing in all the connection settings received from the application.

During Connect(), you 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. Use the obtained values (e.g., hostname, username, password, etc.) to make a connection with your data source by passing them to your relevant API or network protocol.

To establish a connection:

Windows:

- Double-click the TODO #7 message in the Output window. The ULConnection.cpp file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the ULConnection.cpp file located in:

   `[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`

2. Search for TODO #7 or locate the Connect() method.

All Platforms:

1. Look at the code that authenticates the user against your data store using the information provided within the *in_conSettingsRequest* parameter. The sample code uses the helper method Simba::DSI::DSIConnection::*GetRequiredSetting*(). (Note that the map to pass to this helper can be retrieved by calling in_connSettingsRequest.GetSettings())
2. Alternatively, if authentication fails, you can choose to throw an *ErrorException* seeded with DIAG_INVALID_AUTH_SPEC.
3. Save the file.

Linux and macOS only:

- Rebuild your driver:

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source

mk.sh MODE=DEBUG
```

You have now authenticated the user against your data store.

✓ **Day Two Checkpoint – Verify Your Progress** :

- Driver branding is updated with your company/driver name
- DSI_DRIVER_DRIVER_NAME is set to your driver name
- Character encoding settings are configured appropriately
- Log file names are customized for your driver
- Connection settings validation is implemented in UpdateConnectionSettings()
- PromptDialog() is configured (or deferred with return false)
- Connect() authentication logic is implemented
- Driver builds without errors
- Driver connects successfully with proper credentials

Day Three: Metadata Implementation

Today's goal is to return the data used to pass catalog information back to the ODBC-enabled application. Almost all the ODBC-enabled applications require at least the following ODBC catalog functions:

LEARNING OBJECTIVES By the end of this section, you will be able to:

- Understand ODBC catalog functions and their purpose
- Implement metadata sources for SQLGetTypeInfo
- Return catalog, schema, and table type information
- Implement table and column metadata
- Map your data types to standard ODBC SQL types
- SQLGetTypeInfo
- SQLTables (CATALOG_ONLY)
- SQLTables (SCHEMA_ONLY)
- SQLTables (TABLE_TYPE_ONLY)
- SQLTables
- SQLColumns

These catalog functions are represented in the DSI by metadata sources, one for each of the catalog functions.

Create and Return Metadata Sources

TODO #8: Create and return your Metadata Sources

`ULDataEngine::MakeNewMetadataTable`() is responsible for creating the metadata sources to be used to return data to the ODBC-enabled application for the various ODBC catalog functions. Each ODBC catalog function is mapped to a unique `DSIMetadataTableId`, which is then mapped to an underlying MetadataSource that you will implement and return. Each `MetadataSource` instance is responsible for three things:

- Creating a data structure that holds the data relevant for your data store: `Constructor`
- Navigating the structure on a row-by-row basis: `Move`()
- Retrieving data: `GetMetadata`() (See Appendix C: 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`().

Windows:

- In Microsoft Visual Studio, rebuild your solution and double-click the TODO #8 message in the Output window. The ULDataEngine.cpp file opens.
- Locate the MakeNewMetadataTable() method and implement the required metadata sources.
- Click Save.

Linux and macOS:

- Using your preferred text editor or IDE, open the ULDataEngine.cpp file located in:

  `[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`

- Search for TODO #8 or locate the MakeNewMetadataTable() method.
- Implement the required metadata sources.
- Save the file.

Linux and macOS only:

- Rebuild your driver by running the makefile.

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Makefiles

make -f <YourProjectName>.mak debug
```

Handle DSI_TYPE_INFO_METADATA

SQLGetTypeInfo is used by applications to discover data types supported by your driver. The SDK supports all the types listed below but you may want to modify this metadata source if your tables don't support storing all of them or if some of the default metadata differs from our defaults.

The 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 SimbaEngine UltraLight Driver example exposes support for the following subset of the standard ODBC types supported by the SDK:

Type	Type	Type
SQL_BIGINT	SQL_BINARY	SQL_BIT
SQL_CHAR	SQL_DECIMAL	SQL_FLOAT
SQL_DOUBLE	SQL_INTEGER	SQL_LONGVARBINARY
SQL_LONGVARCHAR	SQL_WLONGVARCHAR	SQL_NUMERIC
SQL_REAL	SQL_SMALLINT	SQL_TINYINT

Type	Type	Type
SQL_TYPE_DATE	SQL_TYPE_TIME	SQL_TYPE_TIMESTAMP
SQL_VARBINARY	SQL_VARCHAR	SQL_WCHAR
SQL_WVARCHAR		

3. For your driver, 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.

Handle the Other MetadataSources

The other ODBC catalog functions (including SQLTables (CATALOG_ONLY), SQLTables (TABLE_TYPE_ONLY), SQLTables (SCHEMA_ONLY), SQLTables and SQLColumns) are handled as follows (NOTE: Depending on your datasource, you may choose to support either schemas and/or catalogs. This is up to the developer.):

1. When called with the corresponding metatable ID's, `ULDataEngine::MakeNewMetadataTable`() returns a new instance of one of the following respective 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 fake catalog. This demonstrates how to return a catalog name.

   - **DSITableTypeOnlyMetadataSource**: (default implementation by Simba) 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 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.

2. When called with any other value of `DSIMetadataTableId`, `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 within your data store.

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

✓ **Day Three Checkpoint – Verify Your Progress** :

- ULTypeInfoMetadataSource returns appropriate data types for your data store
- ULCatalogOnlyMetadataSource returns your catalog names
- ULSchemaOnlyMetadataSource returns your schema names
- ULTablesMetadataSource returns your table metadata
- ULColumnsMetadataSource returns your column metadata
- ODBC Test tool can retrieve table list using SQLTables
- ODBC Test tool can retrieve column information using SQLColumns

Day Four: Query Execution

Today's goal is to enable data retrieval from within the driver. We will cover the process of preparing a query, providing parameter information, implementing a query executor, and implementing a result set.

LEARNING OBJECTIVES: By the end of this section, you will be able to:

- Prepare SQL queries for execution against your data source
- Implement an IQueryExecutor to handle query execution
- Handle parameterized queries with input/output parameters
- Implement a DSISimpleResultSet to return query data
- Retrieve and format column data from your data source

Prepare a Query

TODO #9: Prepare a Query

The `ULDataEngine::Prepare`() method takes in a query and is expected to pass it to the underlying SQL enabled datasource for preparation. Once prepared, the method then returns a `ULExecutableOperation` which is used by the engine to return results.

For demonstration purposes, the default 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 therefore 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 `ULExecutableOperation::PopulateParameters`() will populate parameters as described below. In your implementation you would replace this with more sophisticated logic or pass the query to the data source for preparation.

Not all data sources support the notion of preparing a query to the extent that they will be able to have a query plan and produce all the resultant metadata. In these cases, the DSII should make the best effort to determine the number of required query parameters and whether the first result is a rowcount or result set. Precise metadata of the parameters and columns may improve how applications behave with the driver but is not strictly necessary if the data source can only make guesses at this point. Character types are often a safe guess as they support most conversions.

Windows:

- Double-click the TODO #9 message in the Output window. The `ULDataEngine.cpp` file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the ULDriver.cpp file located in:

   `[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`

1. Search for TODO #9 or locate the SetDriverPropertyValues() method.

All Platforms:

1. Look at the `Prepare`() method. This method receives a query string and must return an `IExecutableOperation` implementation.
2. Modify the logic to pass the query to your data source for preparation or implement query parsing appropriate to your needs.
3. Determine whether the query will return a result set or a row count and set the `isSelect` flag appropriately.
4. Check for parameter markers ("?") in the query to determine if the statement is parameterized.
5. Return a new instance of your `IExecutableOperation` implementation (`ULExecutableOperation`).
6. Save the file.

Linux and macOS only:

- Rebuild your driver by running the makefile:

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/So
urce

mk.sh MODE=DEBUG
```

Note: The Prepare() method is called when an application calls SQLPrepare() or SQLExecDirect() (or a JDBC client connecting to the server calls prepare or execute methods on one of its Statement objects). If called for a prepare, the returned IExecutableOperation will be used for subsequent execution calls. A IExecutableOperation object may be executed multiple times with different parameter values (unless created with Simba::DSI::IDataEngine::PrepareMode::ATTEMP_EXEC_DIRECT).

Implement an IExecutableOperation

TODO #10: Implement an IExecutableOperation

The ULExecutableOperation object returned by the ULDataEngine::Prepare() method is an implementation of IExecutableOperation which, as the name suggests, represents a query operation. After preparing a query, an application may execute it multiple times, in which case a single IExecutableOperation would be created by the prepare and would then be used for each execution.

The implementation of ULExecutableOperation is responsible for preparing parameter metadata if needed, and constructing an IExecution instance each time Execute() is called. The IExecution will be responsible for performing the execution, handling parameter values, and constructing IResult instances.

Windows:

- Double-click the TODO #10 message in the Output window. The `ULExecutableOperation.cpp` file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the ULExecutableOperation.cpp file located in:

   `[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`

2. Search for TODO #10 or locate the Execute() method.

All Platforms:

- Look at the `ULExecutableOperation` constructor. This is where the object is initialized with information about the query type.
- Modify the constructor to store any necessary state for your data source connection and query information.
- Review the Execute() method. This method returns the `IExecution` that will perform the execution.
- Save the file.

Linux and macOS only:

- Rebuild your driver by running the makefile:

```
cd [INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source

mk.sh MODE=DEBUG
```

The key methods you need to implement in your ULExecutableOperation are:

- **Execute**(): Creates the ULExecution.
- **GetNumParams**(): Returns the number of parameters in the prepared statement.
- **PopulateParameters**(): Provides metadata about each parameter.

Provide Parameter Information

TODO #11: Provide parameter information

`ULExecutableOperation::PopulateParameters`() method is where parameter information is specified when the application calls SQLPrepare (and _not_ for SQLExecDirect). The default implementation shows how to register input, input/output, and output-only parameters. Modify this method as required to register parameters appropriate for your queries.

Note: This method will only be called if `ULExecutableOperation::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.

Windows:

- Double-click the TODO #11 message in the Output window. The ULExecutableOperation.cpp file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the ULExecutableOperation.cpp file located in:

   `[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`

2. Search for TODO #11.

All Platforms:

1. Look at the `PopulateParameters()` method. This method receives an `IParameterManager` that you use to register each parameter.
2. For each parameter in your query, call `IParameterManager ::RegisterParameter` with the (1-based) parameter number, and use the setter methods on the resulting IParameterSource to provide the metadata, most importantly,
3. Use `IParameterSource::SetParameterType` to tell the SDK if it's an input, output, or input/output parameter
4. Use `IParameterSource::SetSqlType` (along with other setters like SetLength afterwards if needed) or `IParameterSource:: IParameterSource::SetMetadata` to tell the SDK about the parameter's type metadata
5. Save the file.

Linux and macOS only:

- Rebuild your driver by running the makefile:

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source

mk.sh MODE=DEBUG
```

The sample code demonstrates parameter registration as follows:

```
// Register an input-only parameter.
                        IParameterSource* paramSource =
                        in_paramManager.RegisterParameter(1);
                        paramSource->SetSQLType(SQL_VARCHAR);
                        paramSource->SetParameterType(DSI_PARAM_INPUT);
```

```
                    paramSource->SetLength(100);

                    // Register an input-output parameter.
                    paramSource = in_paramManager.RegisterParameter(2);
                    paramSource->SetSQLType(SQL_VARCHAR);
                    paramSource->SetParameterType(DSI_PARAM_INPUT_OUTPUT);
                    paramSource->SetLength(100);

                    // Register an output-only parameter.
                    paramSource = in_paramManager.RegisterParameter(3);
                    paramSource->SetSQLType(SQL_VARCHAR);
                    paramSource->SetParameterType(DSI_PARAM_OUTPUT);
            paramSource->SetLength(100);
```

Implement IExecution

TODO #12: Implement IExecution

The next step is to handle statement execution in `ULExecution::Execute`(). This is responsible for all parts of execution. It uses a loop and switch statement to progress through the various states of execution including handling multiple parameter sets and producing results. The `in_isSelect` parameter passed to the constructor is used here to determine if it should add a simple result set consisting of people's names to `m_results` or a simple row count for each parameter set processed

In your implementation, the parameter handling, retrieval, and storage of the result set can be moved out of this method to wherever necessary.

The `ContinueEecutionResult` returned here is the next result in the current state of execution or sometimes an instruction to the SDK about when ODBC DATA_AT_EXEC parameters should be pushed to the current executionIn the implementation, the `ContinueExecution`()method should begin by serializing parameters (stored in `in_inputParamSetter`) into a form that the data source can consume. Once this has been done then the data source should then be instructed to execute the statement, after which the results should be placed into `the in_outputParamSetIter` parameter.

Once results are ready, the first IResult should be returned. ContinueExecution will be repeatedly called until all results are consumed indicated by returning ExecDone.

Windows:

- Double-click the TODO #12 message in the Output window. The `ULExecution.cpp` file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the ULExecution.cpp file located in:

   `<installdir>/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`

2. Search for TODO #12.

All Platforms:

1. Look at the `ContinueExecution` method. This method is called when the application calls `SQLExecute`() or `SQLExecDirect`().

2. Extract parameter values from `in_inputParamSetter` if the statement has parameters. Use the `GetInputParam`() method to retrieve each parameter value.
3. Send the query (with parameter values) to your data source for execution.
4. If the statement has output parameters, populate them using `in_outputParamSetIter`.
5. Create the appropriate `IResult` objects (`DSISimpleResultSet` for SELECT queries, `DSISimpleRowCountResult` for INSERT/UPDATE/DELETE).
6. Store the results in `m_results` so they can be returned from `ContinueExecution`().
7. Save the file.

Linux and macOS only:

- Rebuild your driver by running the makefile:

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/So
urce

mk.sh MODE=DEBUG
```

Important: The Execute() method may be called multiple times for a single prepared statement, unless created with Simba::DSI::IDataEngine:: PrepareMode::ATTEMP_EXEC_ DIRECT).. Ensure your implementation properly resets state between executions.

Implement an IResult

TODO #13: Implement your IResult

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

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

The 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 example, `ULResultSet::MoveToNextRow`() simply increments a row iterator so this should be replaced in your implementation with code that delegates this to the data source.

The `RetrieveData`() method is where column data is retrieved, so this should also be modified to extract data from the data source. (See Appendix C: Data Retrieval for a brief overview of data retrieval)

Windows:

- Double-click the TODO #13 message in the Output window. The ULResultSet.cpp file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the ULResultSet.cpp file located in: `[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`
2. Search for TODO #13.

All Platforms:

1. Look at the `InitializeColumns`() method. This method defines the metadata for each column in the result set (name, SQL type, precision, scale, nullability).
2. Modify `InitializeColumns`() to define columns that match your data source schema.
3. Look at the `MoveToNextRow`() method. This method advances the cursor to the next row of data.
4. Modify `MoveToNextRow`() to fetch the next row from your data source. Return true if a row was fetched, false if no more rows are available.
5. Look at the `RetrieveData`() method. This method retrieves the value of a specific column in the current row.
6. Modify `RetrieveData`() to extract the column value from your data source and copy it into the provided SqlData buffer.
7. Save the file.

Linux and macOS only:

- Rebuild your driver by running the makefile:

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/So
urce

mk.sh MODE=DEBUG
```

The key methods in ULResultSet are:

- **InitializeColumns**(): Defines the column metadata for the result set. Called during construction.
- **MoveToNextRow**(): Advances the cursor to the next row. Returns true if successful, false if no more rows.
- **RetrieveData**(): Retrieves the value of a column in the current row into a SqlData buffer.
- **GetRowCount**(): Returns the number of rows in the result set (-1 if unknown).
- **HasRowCount**(): Returns true if the row count is known before fetching all rows.

When implementing `RetrieveData`(), you must handle different data types appropriately:

```
void ULResultSet::RetrieveData(
                        simba_uint16 in_column,
                        SqlData* io_data,
                        simba_signed_native in_offset,
                        simba_signed_native in_maxSize)
                        {
                        // For NULL values:
                        if (IsColumnNull(in_column))
                        {
                        io_data->SetNull(true);
                        return false; // No more data for this column
                        }
                        else if (IsInt32(in_column)
                        {
                        // For fixed-length types (INTEGER, SMALLINT, etc.):
                        simba_int32 value = GetIntegerFromDataSource(in_column);
                        in_data->SetNull(false);
                        (static_cast<simba_int32*>(io_data->GetBuffer()))
                         = value;
                        return false; // No more data for this column
                        }
                        else if (IsString(in_column))
                        {
                        // For variable-length types (VARCHAR, VARBINARY):
                        return DSITypeUtilities::OutputWVarCharStringData(
                        GetStringFromDataSource(in_column),
                        in_data,
                        in_offset,
                        in_maxSize);
                        }
                        // Or, if conversion not needed,
                        bool hasMore;
                        std::span<const byte> columnData = GetColumnData(
                        in_column,
                        in_maxSize,
                        in_offset,
                        hasMore);
                        // Helper method which takes care of returning
                         the correct
                        // chunk of data, and no more than SIMBA_UINT32_MAX
                        bytes of it.
                        in_data->SetNull(false);
                        in_data->SetLength(static_cast<simba_uint32>
                        (columnData.size()));
                        memcpy(in_data->GetBuffer(), columnData.data(),
                        columnData.size());
                        return hasMore;
                }
```

Caution: For variable-length data (character and binary types), you must call SetLength() on the SqlData object before calling GetBuffer(). Failure to do so may result in a heap violation. Also ensure you respect the in_maxSize parameter to avoid buffer overflows.

Tip: Day Four Checkpoint – Verify Your Progress:

- ULDataEngine::Prepare() properly handles your operation syntax
- IExecutableOperation is implemented to execute operations against your data source
- Parameter handling is implemented in PopulateParameters() (if applicable)
- Execute() properly serializes parameters and executes statements
- DSISimpleResultSet returns data rows from your data source
- SELECT queries return proper result sets in ODBC Test tool
- Non-SELECT queries return appropriate row counts

Day Five: Productization

Today's goal is to start productizing the driver. Additionally, you can also start localizing the driver error messages. Refer to the SimbaEngine Developer Guide for more details.

LEARNING OBJECTIVES By the end of this section, you will be able to:

- Configure custom error messages for your driver
- Set vendor branding for all error messages
- Rename files and classes for production deployment
- Create a driver configuration dialog
- Finalize the driver for customer deployment

Configure Error Messages

TODO #14: Register Messages xml file for handling by DSIMessageSource

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

Windows:

- Double-click the TODO #14 message in the Output window. The `DSIMessageSource.cpp` file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the `ULResultSet.cpp` file located in:

   `[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Source/`

2. Search for TODO #14 and open the file.

All Platforms:

1. Rename the `ULMessages.xml` file to something appropriate to your data store.
2. Update the line associated with the TODO to match the new name of the file.
3. Open the renamed file and change all instances of the following items:
   a. The letters `UL` to a two-letter abbreviation of your choice in each `<Error>` element
   b. The word `UltraLight` to a name relating to your driver
4. When you are done, you should revisit each exception thrown within your DSI implementation and change the parameters to match as well. This will rebrand your converted SimbaEngine UltraLight Driver for your organization.
5. Save the file.

Linux and macOS only:

- Rebuild your driver by running the makefile:

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/So
urce

mk.sh MODE=DEBUG
```

TODO #15: Set the vendor name, which will be prepended to error messages

The vendor name is prepended to all error messages that are visible to applications. The default vendor name is Simba. To set the vendor name:

Windows:

- Double-click the TODO #15 message in the Output window. The file opens.

Linux and macOS:

1. Using your preferred text editor or IDE, open the a file located in:

   ```
   [INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/Sourc
   e/
   ```

2. Search for TODO #15 and open the file.

All Platforms:

1. Set the vendor name string as shown in the commented example code.
2. Save the file.

Linux and macOS only:

- Rebuild your driver by running the makefile:## Finishing Touches

```
cd
[INSTALLDIR]/SimbaEngineSDK/10.4/Examples/Source/<YourProjectName>/So
urce

mk.sh MODE=DEBUG
```

Finishing Touches

You are now done with all of the TODO's in the project. However, there are still a couple of final steps before you have a fully functioning driver:

- Rename all files and classes in the project to have the two-letter abbreviation chosen as part of TODO #14.

- Create a driver configuration dialog. This dialog is presented to the user when they use the ODBC Data Source Administrator to create a new ODBC DSN or configure an existing one. The C++ SimbaEngine UltraLight Driver project contains an example ODBC configuration dialog that you can look at, as an example.

- To see the driver configuration dialog that you created, run the ODBC Data Source Administrator, open the Control Panel, select Administrative Tools, and then select Data Sources (ODBC).

You are now done with all the TODO's in the project. You have created your own, custom ODBC driver using SimbaEngine by modifying and customizing the UltraLight sample driver. Now, you have a read-only driver that connects to your data store.

Reference

This section contains additional technical information that you may find useful when developing your ODBC driver using the SimbaEngine SDK.

Appendix A: ODBC Data Source Administrator on Windows 32-Bit vs. 64-Bit

On a 64-bit Windows system, you can execute 64-bit and 32-bit applications transparently, which is beneficial because many applications are still 32-bit. Microsoft Excel 2010 and later versions are available in both 64-bit and 32-bit versions, so it is highly likely that you will encounter 32-bit applications running on 64-bit systems.

Understanding Driver and Application Bitness

It is critical to understand that 64-bit applications can only load 64-bit drivers and 32-bit applications can only load 32-bit drivers. In a single running process, all of the code must be either 64-bit or 32-bit. This architectural constraint has significant implications for driver development and deployment.

Application Type	Required Driver Type	ODBC Administrator
64-bit application	64-bit driver only	64-bit ODBC Administrator
32-bit application on 64-bit Windows	32-bit driver only	32-bit ODBC Administrator
32-bit application on 32-bit Windows	32-bit driver only	32-bit ODBC Administrator

Accessing the ODBC Data Source Administrator

On a 64-bit Windows system, the ODBC Data Source Administrator that you access through the Control Panel can only be used to configure data sources for 64-bit applications. However, the 32-bit version of the ODBC Data Source Administrator must be used to configure data sources for 32-bit applications. This is the source of many confusing problems where what appears to be a perfectly configured ODBC DSN does not work because it is loading the wrong kind of driver.

To access the ODBC Data Source Administrators:

Administrator Version	Access Method	Executable Path
64-bit	Control Panel > Administrative Tools > ODBC Data Sources	C:\Windows\System32\odbcad32.exe
32-bit	Must run executable directly	C:\Windows\SysWOW64\odbcad32.exe

Tip: ✓ For convenience during development, create desktop shortcuts to both versions of the ODBC Data Source Administrator. This allows quick access when testing both 32-bit and 64-bit configurations.

Because of this architectural separation, it is very important when using 64-bit Windows that you configure 32-bit and 64-bit drivers using the correct version of the ODBC Data Source Administrator for each. Using the wrong administrator will result in DSNs that are invisible to your applications.

Common Issues and Solutions

Issue	Cause	Solution
DSN not visible in application	DSN created with wrong bitness administrator	Recreate DSN using correct administrator version
Driver not found error	Application bitness does not match driver bitness	Install correct driver version (32-bit or 64-bit)
Configuration changes not taking effect	Editing wrong registry location	Verify registry path matches driver bitness
ODBC Test tool does not show DSN in applications (such as ODBCTest or Excel)	Using wrong ODBC Test tool version	Use 32-bit or 64-bit ODBC Test tool to match DSN

Appendix B: Windows Registry Structure for ODBC Drivers

As noted previously, the 32-bit and 64-bit drivers must remain clearly separated because you cannot use a 32-bit driver from a 64-bit application or vice versa. The 32-bit and 64-bit ODBC drivers are installed and data source names are created in different areas of the Windows registry.

Registry Structure Overview

The ODBC Driver Manager uses specific registry keys to locate drivers and data source configurations. Understanding this structure is essential for proper driver installation and troubleshooting.

Configuration	Registry Root Path
64-bit drivers on 64-bit Windows	HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\
32-bit drivers on 64-bit Windows	HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\ODBC\
32-bit drivers on 32-bit Windows	HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\

32-Bit Drivers on 64-Bit Windows

The 32-bit applications and drivers use a section of the registry that is separate from the 64-bit applications and drivers. This separation is handled automatically by the Windows WOW64 (Windows 32-bit on Windows 64-bit) subsystem. Note that from the point of view of a 32-bit application on a 64-bit machine, 32-bit data sources look exactly like they do on a 32-bit machine.

Data Source Names (ODBC.INI):

To connect your driver to your database, the 32-bit ODBC Driver Manager on 64-bit Windows uses Data Source Name registry keys in:

HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\ODBC\ODBC.INI

This location contains:

- ODBC Data Sources: A subkey listing all configured DSN names and their associated drivers
- Individual DSN subkeys: Each DSN has its own subkey containing connection parameters (Driver, Description, Server, Database, etc.)

Driver Definitions (ODBCINST.INI):

To define each driver and its setup location, the 32-bit ODBC Driver Manager on 64-bit Windows uses registry keys created in:

HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\ODBC\ODBCINST.INI

This location contains:

- ODBC Drivers: A subkey listing all installed driver names with "Installed" status
- Individual Driver subkeys: Each driver has its own subkey containing Driver path, Setup DLL path, and other configuration

64-Bit Drivers on 64-Bit Windows

64-bit drivers use the standard registry paths without the WOW6432Node redirection.

Data Source Names (ODBC.INI):

To connect your driver to your database, the 64-bit ODBC Driver Manager on 64-bit Windows uses Data Source Name registry keys in:

HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI

Driver Definitions (ODBCINST.INI):

To define each driver and its setup location, the 64-bit ODBC Driver Manager on 64-bit Windows uses registry keys created in:

HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI

Required Registry Keys for Driver Installation

When creating an installer for your ODBC driver, you must create the following registry entries:

1. Register the Driver in ODBC Drivers list:

   [HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers]
   "YourDriverName"="Installed"

2. Create Driver Definition:

   [HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\YourDriverName]
   "Driver"="C:\\Path\\To\\YourDriver.dll" "Setup"="C:\\Path\\To\\YourDriverSetup.dll"
   "APILevel"="2" "ConnectFunctions"="YYY" "DriverODBCVer"="03.80" "FileUsage"="0"
   "SQLLevel"="1"

3. Register the DSN in ODBC Data Sources list:

   [HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\ODBC Data Sources]
   "YourDSNName"="YourDriverName"

4. Create DSN Definition:

   [HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\YourDSNName]
   "Driver"="C:\\Path\\To\\YourDriver.dll" "Description"="Description of your data source"
   "Server"="localhost" "Database"="mydb" "UID"="" "PWD"=""

   > **Important:** When writing paths in .reg files, you must use double backslashes (\\) as the backslash is an escape character. When writing directly to the registry via code, use single backslashes. We do not recommend storing unencrypted passwords in the DSN.

Appendix C: Data Retrieval Implementation

In the Data Store Interface (DSI), data retrieval is one of the most critical operations. This appendix provides detailed information on implementing data retrieval in your ODBC driver.

Data Retrieval Methods

The following two methods perform the task of retrieving data from your data store:

- Each MetadataSource implementation's GetMetadata() method - for catalog function results
- IResult::RetrieveData() method - for query result set data

Both methods provide a way to uniquely identify a column within the current row. For MetadataSource, SimbaEngine passes in a unique column tag (see DSIOutputMetadataColumnTag). For IResult, SimbaEngine passes in the column index (0-based).

Method Parameters

Both data retrieval methods accept the following parameters:

1. in_data

   The SqlData object into which you must copy the value of your cell. This class is a wrapper around a buffer managed by the Simba SQL Engine. Key points about SqlData:

   a. Call GetBuffer() to access the raw buffer pointer for writing data
   b. By default, for fixed-length types, the buffer is allocated when the SqlData is created and remains fixed within its lifetime, but for variable-length types, the buffer is allocated via SqlData::SetLength.
   c. Note: There's an optional mode for SqlData where the buffer is managed externally from the SqlData object, to reduce copies. For more information, search for the definition of TDW_ BUFFER_ATTACHED in the SDK header files.
   d. Data must be formatted according to the SQL type (see SQL Type to C Type mapping below)
   e. For NULL values, call SetNull(true) instead of writing data

2. in_offset
   a. Fixed length columns should ignore this value, but for variable length columns (i.e character/binary types), this is the offset (in bytes) into the value to begin returning data from. The SDK will always begin at 0, and increment this offset exactly by the number of bytes returned in the previous call for this column for the current row. Note that if the DSII overrides

the value of DSI_RETRIEVE_DATA_ORDER_RESTRICTION to include DSI_RETRIEVE_DATA_ANY_ORDER, it may restart at offset 0 without moving to the next row if the application attempts to re-fetch the column.

3. in_maxSize (simba_signed_native)
4. The maximum number of bytes that can be copied into the in_data buffer, or -1 (i.e RETRIEVE_ALL_DATA) if there is no limit and all remaining data for the column should be returned. Fixed length columns should ignore this value, but for variable-length columns (i.e. character/binary types), you should try to limit the amount of data returned in the SqlData to what is requested for optimal performance

SQL Type to C Type Mapping

When retrieving data, the buffer type in SqlData corresponds to the column's SQL type. The following table shows the mapping between SQL types and the C types used in the buffer:

SQL Type	C Type	Size (bytes)
SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR	simba_char* (null-terminated)	Variable
SQL_WCHAR, SQL_WVARCHAR, SQL_WLONGVARCHAR	simba_wchar* (null-terminated)	Variable
SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY	simba_byte*	Variable
SQL_BIT	simba_byte	1
SQL_TINYINT	simba_uint8	1
SQL_SMALLINT	simba_int16	2
SQL_INTEGER	simba_int32	4
SQL_BIGINT	simba_int64	8
SQL_REAL	simba_float32	4
SQL_FLOAT, SQL_DOUBLE	simba_double64	8
SQL_NUMERIC, SQL_DECIMAL	SQL_NUMERIC_STRUCT	19
SQL_TYPE_DATE	SQL_DATE_STRUCT	6
SQL_TYPE_TIME	SQL_TIME_STRUCT	6
SQL_TYPE_TIMESTAMP	SQL_TIMESTAMP_STRUCT	16

Implementation Examples

The RetrieveData implementation in TODO #13: Implement your IResult demonstrates the handling of fixed-length, variable-length, and null values within an IResult implementationPerformance Considerations

When implementing data retrieval, consider the following performance optimizations:

- Buffer data locally: Fetch multiple rows at once from your data source and cache them to reduce round-trips.
- Lazy loading for LOB data: For large objects (LONGVARCHAR, LONGVARBINARY), consider fetching the data only when RetrieveData() is called for that column.
- Use appropriate fetch sizes: When communicating with your data source, balance memory usage against network round-trips.
- Avoid unnecessary conversions: If your data source stores data in a format compatible with ODBC types, avoid intermediate conversions.
- Handle chunked retrieval efficiently: For large data, track the current position to avoid re-reading from the beginning on each call.

Appendix D: Configuring the C++ Server

SimbaEngine supports building ODBC drivers as standalone servers in addition to traditional DLL-based drivers. This appendix covers the configuration options for server-based deployments.

Server Architecture Overview

In a server configuration, the ODBC driver runs as a separate process (or on a separate machine) from the client application. This architecture provides several benefits:

- Centralized data access: Multiple clients can connect through a single server instance. Access control & query rewriting can happen here to limit what a client can do.
- Resource isolation: Driver crashes do not affect client applications.
- Cross-platform connectivity: A server on Linux can serve Windows clients and vice versa.
- Simplified deployment: Only the client component needs to be installed on end-user machines.
- Connection pooling: The server can manage and reuse connections to the underlying data source.

Server Configuration Settings

The SimbaEngine server reads configuration from an INI file. The following settings are available:

Setting	Description	Default Value
ListenPort	TCP port number for client connections	10400
MaxConnections	Maximum concurrent client connections	100
LIVENESSCHECKINTERVAL	Seconds before the server attempts to validate a connection is still live	3600 (1 hour)
IdleTimeout	Seconds before idle connections (those not submitting any new requests or processing existing ones) are closed	86400 (24 hours)

Setting	Description	Default Value
USESSL	Enable TLS/SSL encryption: ■ 0/Disabled/N – off (allows only plaintext connections) ■ 1/Enabled/Y – on (allows all connections) ■ Required – allows only encrypted connections	0
SSLCertFile	Path to SSL certificate file	(none)
SSLKeyFile	Path to SSL private key file	(none)
LogLevel	Logging verbosity (0-6)	0 (Disabled)
LogPath	Directory for server log files	(none)

Example server configuration file (simbaserver.ini):

```
[Server]
                        ListenPort=10400
                        MaxConnections=100
                        IdleTimeout=3600
                        EnableSSL=1
                        SSLCertFile=/etc/ssl/certs/server.crt
                        SSLKeyFile=/etc/ssl/private/server.key

                        [Logging]
                        LogLevel=3
                        LogPath=/var/log/simbaserver

                        [DataSource]
                        ConnectionPoolSize=20
                        ConnectionTimeout=30
```

Client Connection String Parameters

When connecting to a SimbaEngine server, clients use the following connection string parameters:

Parameter	Description	Example
SERVERLIST	List of server hostname or IP address and port pairs (separated by commas). The client attempts a connection to each and chooses one that succeeds.	■ SERVERLIST=192.168.1.100 10400 ■ SERVERLIST=first.com 12345,second.com 12346
UseSSL	Enable TLS/SSL encryption: ■ 0/Disabled/N – does not attempt to enable SSL ■ 1/Enabled/Y – attempts to enable	UseSSL=1

Parameter	Description	Example
	SSL but tolerates the server not supporting it ■ Required – attempts to enable SSL and drops the connection if the server does not support it	
Timeout	Connection timeout in seconds	Timeout=30

Example connection string:

```
Driver={YourDriverName};ServerList=dataserver.example.com
10400;UseSSL=1;UID=analyst;PWD=secret;Timeout=30
```

Glossary

Term	Definition
DSI	Data Store Interface – The API that defines operations needed to access a data store
DSII	Data Store Interface Implementation – Your custom code that implements the DSI
DSN	Data Source Name – A named configuration for connecting to a specific data source
IExecutableOperation	Interface for preparing and executing SQL statements
MDAC	Microsoft Data Access Components – Microsoft's data access framework
Metadata	Data that describes the structure of your data (tables, columns, types)
MetadataSource	Classes that return catalog information (tables, columns, types)
ODBC	Open Database Connectivity – A standard API for database access
Result Set	The data returned from executing a SELECT query
Row Count	The number of rows affected by a non-SELECT query
SDK	Software Development Kit – The SimbaEngine development package
SQL	Structured Query Language – Standard language for database queries
UltraLight	The sample DSI implementation included with SimbaEngine

Third Party Licenses

The SimbaEngine SDK includes software developed by third parties. This section provides license information for the third-party components included in the SDK. For complete license information, please refer to the third-party-licenses.txt file that is packaged with the SimbaEngine software.

Note: The third-party-licenses.txt file is located in the [INSTALLDIR]\SimbaEngineSDK\10.4\Docs directory and contains the complete text of all third-party licenses applicable to the SDK.

fast_float License

The SimbaEngine SDK uses the fast_float library for high-performance floating-point number parsing.

Apache License, Version 2.0

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at:

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

The fast_float library is also available under the MIT License. The MIT License text is as follows:

MIT License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

ICU License (ICU 1.8.1 and later)

The SimbaEngine SDK uses the International Components for Unicode (ICU) library for Unicode support and internationalization services.

COPYRIGHT AND PERMISSION NOTICE

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

OpenSSL License

The SimbaEngine SDK may use OpenSSL for SSL/TLS encryption support in server configurations.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
- The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org.
- Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project.
- Redistributions of any form whatsoever must retain the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Original SSLeay License

The OpenSSL toolkit includes cryptographic software written by Eric Young (eay@cryptsoft.com) and software written by Tim Hudson (tjh@cryptsoft.com).

Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved.

This package is an SSL implementation written by Eric Young (eay@cryptsoft.com). The implementation was written so as to conform with Netscape's SSL.

This library is free for commercial and non-commercial use as long as the following conditions are adhered to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used. This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

libexpat License

The SimbaEngine SDK uses the Expat XML parser library (libexpat) for XML parsing functionality.

MIT License

Copyright © 1998-2026 Thai Open Source Software Center Ltd and Clark Cooper

Copyright © 2001-2026 Expat maintainers

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Third-Party Components Summary

The following table summarizes the third-party components included in the SimbaEngine SDK:

Component	Purpose	License Type
fast_float	High-performance floating-point parsing	Apache 2.0 / MIT (dual license)
ICU	Unicode and internationalization support	ICU License
OpenSSL	SSL/TLS encryption	OpenSSL License + SSLeay License
libexpat	XML parsing	MIT License

Important: When distributing applications built with the SimbaEngine SDK, you must include appropriate license notices and attributions for any third-party components that are included in your distribution. Refer to the third-party-licenses.txt file for complete license texts and attribution requirements.

For questions regarding third-party licensing or compliance, please contact insightsoftware at www.insightsoftware.com.