insightsoftware

Simba SDK

Developing Connectors for SQL-Capable Data Stores

Version 10.3

August 2024

Copyright

Contact Us

insightsoftware

www.insightsoftware.com

About This Guide

Purpose

This guide explains how to use the Simba SDK to document what SQL grammar the engine supports.

Audience

The guide is intended for developers who have created a connector with the Simba SDK. This guide is also intended for end users of the Simba SDK.

Knowledge Prerequisites

To use the Simba SDK, the following knowledge is helpful:

- Familiarity with the platform on which you are using the Simba SDK.
- Ability to use the data store to which the Simba SDK is connecting.
- An understanding of the role of ODBC or JDBC technologies and driver managers in connecting to a data store.
- Experience creating and configuring ODBC or JDBC connections.

Variables Used in this Document

The following variables are used in this document:

Variable	Description
[DRIVER_NAME]	The name of your connector, as used in Windows registry keys and names of configuration files.
[INSTALL_DIR]	Installation directory for the Simba SDK.

Contents

Introducing the Simba SDK

The Simba Software Development Kit (SDK) is a collection of database access tools packaged in a flexible, reusable set of components. These components are used to create custom database connectors for any data store, even if the data store is not SQL-capable. Connectors can be built to access both local and remote data stores.

This guide introduces the components of the Simba SDK and explains how you can use them to create custom connectors for ODBC, JDBC, OLE DB and ADO.net applications.

Note:

This guide explains how to build a connector for data stores that support SQL. If you want to build a connector for data stores that do not support SQL, see Developing Connectors for Data Stores Without SQL.

You may find the HTML version of this guide easier to use. See Developing Connectors for SQL-capable Data Stores.

Creating a Custom Connector with the Simba SDK

The components of the Simba SDK implement all the required functionality of ODBC, JDBC, OLE DB, and ADO.net, as well as handling session management, state management, data conversion, and error checking. These components provide an abstraction layer to insulate your underlying connector functionality from any changes to data access standards. By basing a custom connector on the Simba SDK, you can leverage the experience of leaders in data connectivity.

For data stores that do not support SQL, the Simba SDK provides an SQL parser and an execution engine to translate between SQL commands and your custom datastore API.

For data stores requiring remote deployment, the Simba SDK allows you to re-build your existing connector into a server for a client/ server deployment. This allows you to build your connector as a server that reside near data source, then deploy an ODBC or a JDBC client that handles communication with the end user's application. For more information about client-server deployment, see the *SimbaClientServer User Guide* at http://www.simba.com/resources/sdk/documentation/.

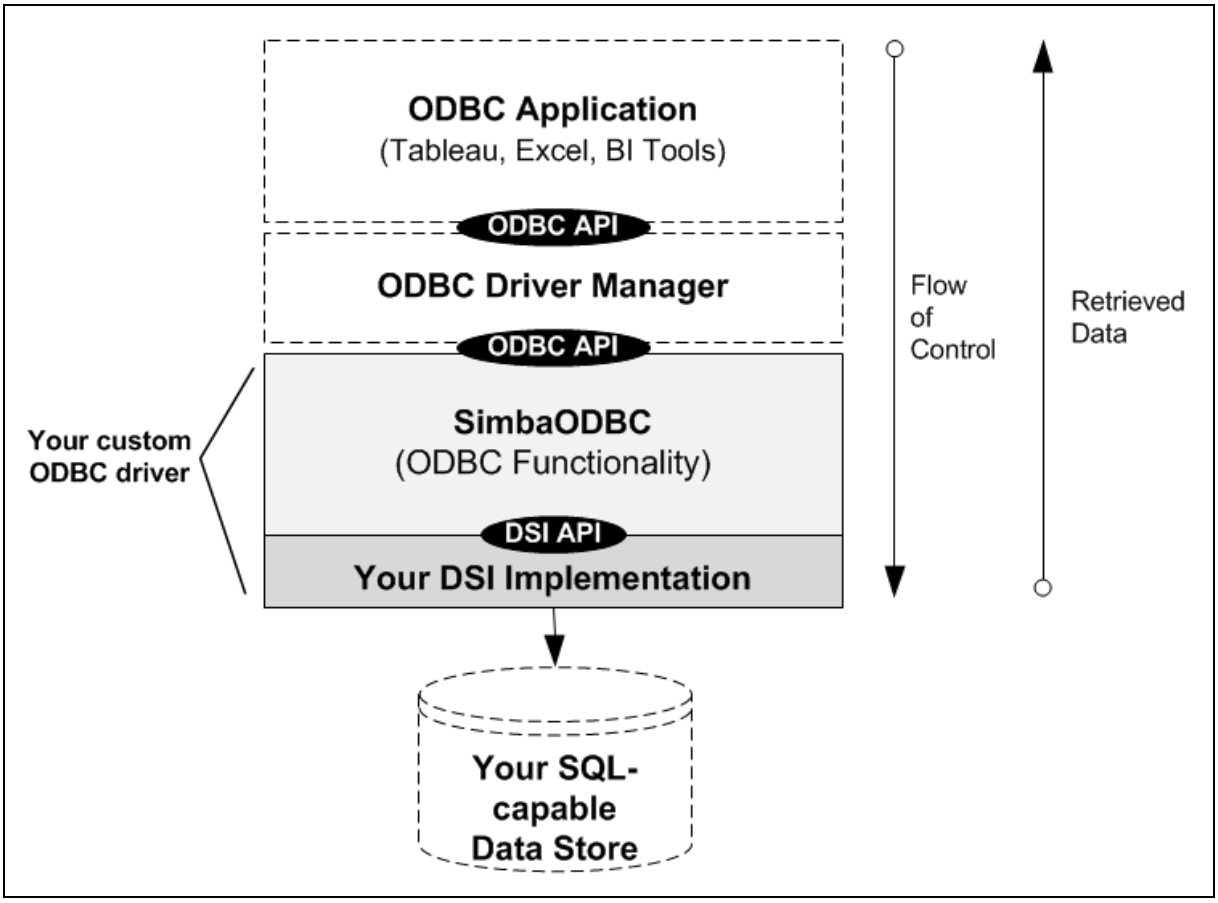
Data Store Interface Implementation (DSII)

To write a custom connector using the Simba SDK, you write a component called the "DSI implementation" to access your data store. You then link this component with the Simba SDK components, which takes care of meeting the data access standards, and optionally converting SQL commands to commands that your data store can understand. The result is a shared object: a `.dylib`, `.jar`, or `.dll`, `.so` file, depending on your development platform. Applications, such as Tableau or Microsoft Excel, use this shared object to access your data store, even if your data store is not SQL-enabled.

Example - Build an ODBC Connector for a SQL-Capable Data Store

The easiest custom ODBC connector you can build with the Simba SDK is a standalone connector connecting to an SQL-capable data store. In this configuration, the application (such as Tableau or Excel) creates SQL queries and sends them to the ODBC connector. The ODBC connector can choose to modify these queries, then sends them to the data store. The data store executes the SQL queries and creates a result set. Finally, the ODBC connector moves the result set from the data store back to the application.

This flow of control is illustrated below:



Note:

The Simba SDK provides a similar solution for JDBC, OLE DB, and ADO.net applications.

The following sections describe the components shown in the above diagram.

SimbaODBC Component

For data stores that are SQL-capable, your custom ODBC connector is composed of the SimbaODBC component and your DSI implementation. The SimbaODBC component implements most of the connector functionality, including:

- session and statement management
- abstracting and implementing the low-level requirements of the ODBC API
- error checking

Note:

When changes are made to the ODBC API, or when applications change how they use the ODBC API, the Simba SDK incorporates these changes transparently. As a result, connectors based on the Simba SDK can handle these changes without code rewrites.

The Data Store Interface (DSI)

The data store interface, or DSI, defines a generic view of an SQL database that is independent of the data access standards (ODBC, JDBC, ADO.NET and OLE DB). The Simba SDK translates the ODBC, JDBC, ADO.NET and OLE DB interfaces to the DSI in C++, Java, or C#. By writing code to map from the DSI to your data store, you are creating a connector that can use one of these standard interfaces.

Note:

- The DSI API is object-oriented and simpler to use than the industry-standard interfaces, making it easier to translate standard APIs to your custom data store.
- The DSI API provides a consistent API for all the standards it supports: ODBC, JDBC, ADO.NET or OLE DB. This makes creating connectors for different standards much easier, because you can re-use your knowledge.

Your DSI Implementation (DSII)

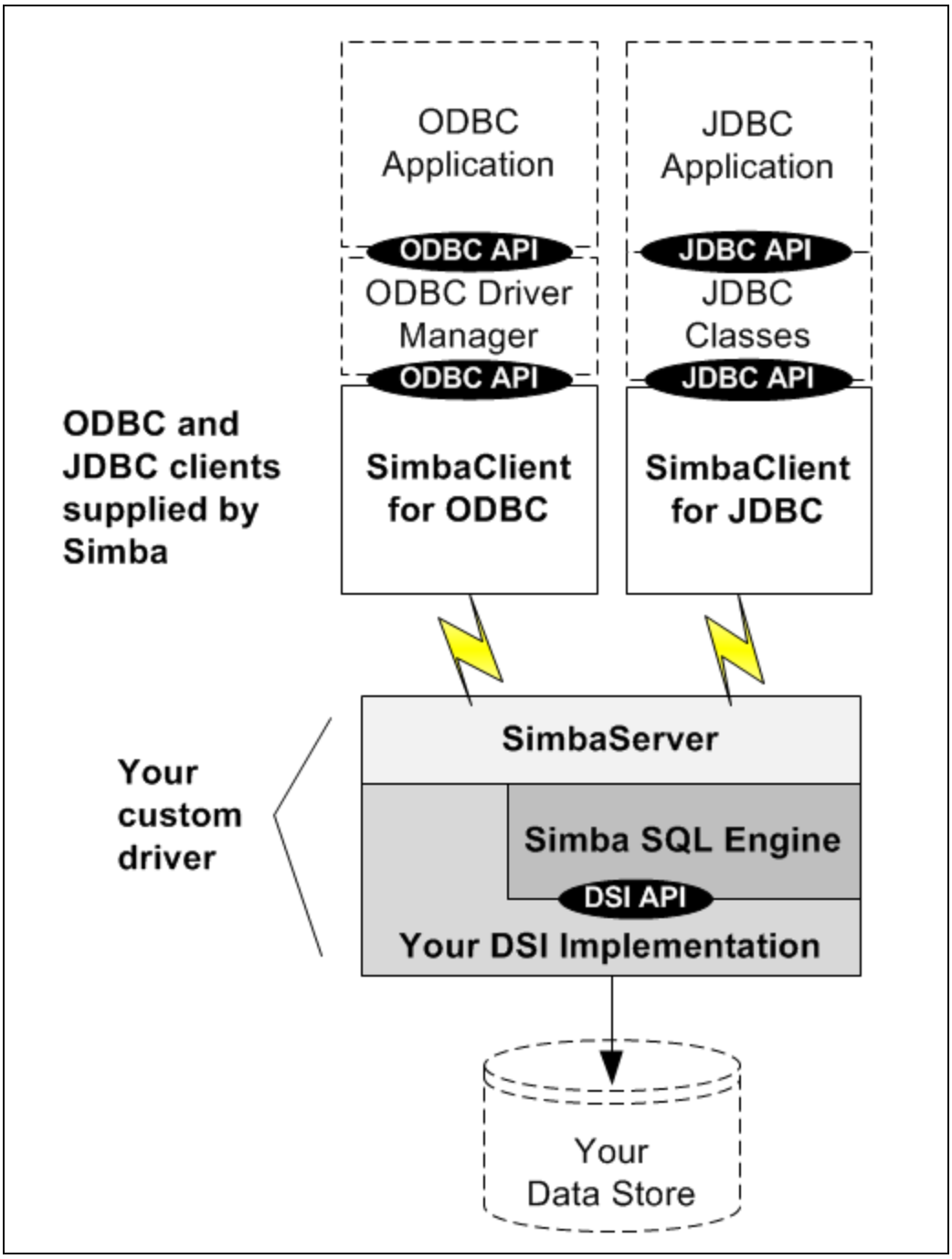
The SimbaODBC component uses the data store interface, or DSI, to communicate with the your DSI implementation. The DSI interface is common to all Simba SDK components that communicate with customer code. You write your DSI implementation (DSII) to connect directly to your data store and translate its interface to the DSI API.

Note:

Every DSII is custom designed for a specific data store and that data store's interface.

Example - Build a Client/Server Solution

Once you have created a DSI implementation and built a custom connector, either for a SQL-enabled or non-SQL-enabled data store, you can rebuild your DSI implementation into a client/server solution. You can do this without making any changes to the code - simply link your DSI implementation to the Simba Server to provide remote data access:



The following sections describe the components shown in the above diagram.

Simba Client/Server protocol

The Simba Client/Server protocol is a network protocol that works on any network to provide remote access to a DSI implementation. Simba Server translates the Simba Client/Server protocol to the DSI API.

Note:

- A client/server deployment lets you locate your custom connector close to the data store, while the client applications are located with your users.
- Both the ODBC and the JDBC client can talk to the same SimbaServer. That means you can write one custom connector, built it as a server, then use it to service SQL requests from both ODBC and JDBC applications.

SimbaClient for ODBC and Simba Client for JDBC

The ODBC and JDBC clients are shared objects provided by Simba. These clients use the Simba Client/Server protocol to handle communication between the application and Simba Server.

Related Topics

Simba SDK Usage Scenarios

Build a Connector in 5 Days

Simba SDK FAQ

Implementation Options

You can use the Simba SDK to build custom connectors for ODBC, JDBC, OLE DB, and ADO.Net applications. Depending on the interface standard that your connector supports, you can develop the connector in C++, Java, or C#.

The Simba SDK provides many different implementation options for developing your custom connector. For example, you can develop an ODBC connector in C++ using the DSI API. You can also develop an ODBC connector in Java using the Java DSI API and the JNI bridge. Or, you can develop a custom JDBC connector for data stores that do not support SQL, and implement the connector for either a local or a client-server deployment.

Note:

This guide explains how to build a connector for data stores that support SQL. If you want to build a connector for data stores that do not support SQL, see Developing Connectors for Data Stores Without SQL.

You may find the HTML version of this guide easier to use. See Developing Connectors for SQL-capable Data Stores.

The following table shows the possible types of custom connectors you can build with the Simba SDK, and the components and APIs required for each. The table includes options for local and remote (client/server) deployments, and for SQL-enabled and non-SQL-enabled data stores. The *Sample Connector(s)* column lists the sample connector(s) that provide a working example of your chosen implementation option.

Note:

- Every connector, except for those written in C#, is supported on Windows, Unix/Linux, and macOS. C# is supported on Windows.
- The sample connectors are included with the Simba SDK in the folder `C:\Simba Technologies\SimbaEngineSDK\10.0\Examples\Source`.

Connector Type	Language	Data Store Type	Sample Connector (s)	Simba SDK Component(s)
Custom ODBC connector	C++	SQL, Local	Ultralight	DSI API
Custom ODBC connector	C++	SQL, Remote	Ultralight + SimbaServer	DSI API
Custom ODBC connector	Java	SQL, Local	JavaUltraLight	Java DSI API + JNI DSI
Custom ODBC connector	Java	SQL, Remote	JavaUltraLight + SimbaServer	Java DSI API + JNI DSI
Custom ODBC connector	C#	SQL, Local	DotNetUltraLight	.NET DSI API + CLI DSI
Custom ODBC drive	C#	SQL, Remote	DotNetUltraLight + SimbaServer	.NET DSI API + CLI DSI
Custom JDBC connector	Java	SQL, Local	JavaUltraLight	Java DSI API
Custom JDBC connector	Java	SQL, Remote	JavaUltraLight + SimbaServer	Java DSI API + JNI DSI
Custom JDBC connector	Java	Not SQL capable, Local	JavaQuickJson	Java DSI API
Custom ADO.NET connector	C#	SQL, Local	DotNetUltraLight	.NET DSI API

The following section provides more details about the information in the table above.

Options for Programming Languages

The programming language you use to write the DSII depends partly on the interface standard you need to support. The supported combinations of programming language and interface standard are shown in the table above.

Example:

- To write a JDBC connector that is deployed locally, you must write the DSII in Java.

- To write an ODBC connector that is deployed locally, you can write the DSII in C++, Java, or C#. If you write the DSI in Java, you need to link with a JNI bridge. If you write the DSI in C#, you need to link with a CLI bridge.

Programming Languages for ODBC applications

To build a local connector for ODBC applications, you can write your DSII in the following languages:

- C++ (the most common choice)
- C# with a CLI bridge
- Java with a JNI bridge

Programming Language for JDBC Applications

To build a local connector for JDBC applications, you must write your DSII in Java. Or, you can deploy the JDBC client to support the JDBC applications and implement the SimbaServer in Java, C++, or C#.

Programming Language for ADO.NET Applications

To build a local connector for ADO.NET applications, you must write your DSII in C#.

Supported Combination of Components

This section explains the different ways you can leverage the Simba SDK components in each of the supported programming languages.

C++ Development

For C++ connector development, you have the following options:

- Use the DSI API, build as an ODBC connector (connected locally to your data store) and link your DSII to SimbaODBC.
- Build as a SimbaServer connector, supporting remote connections from SimbaClients for JDBC and ODBC. Link your C++ DSII upwards to SimbaServer via the DSI API.

In the above cases, you can link against the C++ SQLEngine to access non-relational data stores.

Java Development

For Java connector development, you have the following options:

- Use the Java DSI API, build as a JDBC connector (connected locally to your data store) and link your DSII with SimbaJDBC.
- Build as an ODBC connector (connected locally to your data store) using the Java DSI API and link via the C++ to Java Bridge to SimbaODBC.
- Build as a SimbaServer connector, supporting remote connections from the JDBC and ODBC clients. Link your Java DSII upward via the Java DSI API and C++ to Java Bridge to SimbaServer.

In the above cases, you can link to the Java SQLEngine to access non-relational data stores.

C# Development

For C# development, you have the following options:

- Use the DotNet DSI API, build as an ADO.NET connector (connected locally to your data store) and link your DSII with Simba.NET.
- Use the DotNet DSI API, build as an ODBC connector (connected locally to your data store) and link via the C++ to C# Bridge to SimbaODBC.
- Build as a SimbaServer connector, supporting remote connections from the JDBC and ODBC clients. Link your DotNet DSII upward via the DotNet DSI API and C++ to C# Bridge to SimbaServer.

Options for Deployment

The Simba SDK provides you a number of different optional components for building and deploying a custom connector for a wide variety of solutions.

Local Deployments

Local deployments are typically used in the following scenarios:

- Client applications access a database that runs on each user's machine.
  For example, an ODBC connector might support a client management database where each user performs analysis of their own, local data.
- You have already configured your database for network access and some component of your software is already installed on user machines. Your new connector will allow other, general-purpose client applications to access the same connection to your database that your own client application uses.
- You are in the early stages of testing your connector and as a developer, you are accessing a local instance of your database. You will eventually change the compilation options to link to the SimbaServer libraries, but there will be no changes needed to your DSI implementation to do this.

Remote (Client/Server) Deployments

Client-Server deployments are best when software runs on a server and users access it from their own machines. Your custom connector, using SimbaServer, runs on the network server. SimbaClient is installed on user machines to allow applications such as Excel and Tableau to access your remote data store.

Related Topics

"Introducing the Simba SDK" on page 8

Simba SDK FAQ

Library Components

This section introduces the components comprising the Simba SDK.

Note:

This guide explains how to build a connector for data stores that support SQL. If you want to build a connector for data stores that do not support SQL, see Developing Connectors for Data Stores Without SQL.

You may find the HTML version of this guide easier to use. See Developing Connectors for SQL-capable Data Stores.

SimbaODBC (the "C++ SDK")

SimbaODBC provides a complete ODBC 3.80 interface and all of the processing required to meet the ODBC 3.80 specification. It is the connection between your custom DSI implementation and ODBC applications such as Tableau and Microsoft Excel.

Your custom ODBC connector is composed of the SimbaODBC component and your DSI implementation. The SimbaODBC component implements most of the connector functionality, including:

- session and statement management
- abstracting and implementing the low-level requirements of the ODBC API
- error checking

Tip:

When changes are made to the ODBC API or the way the standard is used by applications, Simba incorporates these changes in a manner that is transparent to your DSI implementation.

For information about the Simba SDK C++ API method calls, see the Simba SDK C++ API Reference at https://www.simba.com/docs/SDK/SimbaEngine_C++_API_Reference.

Simba OLE DB

This component is part of the C++ SDK. SimbaOLEDB provides interfaces and all the processing required to meet the OLE DB specification. It is the connection between your custom DSI implementation and common OLE DB reporting applications such as Microsoft SQL Server Analysis Services.

SimbaJDBC (the "Java SDK")

SimbaJDBC provides complete interfaces for JDBC 4.0, JDBC 4.1, and JDBC 4.2, as well as all of the processing required to meet these specifications. It is the connection between your custom DSI implementation and common JDBC reporting applications.

For information about the Simba SDK Java API method calls, see the SimbaSDK Java API Reference at https://www.simba.com/docs/SDK/SimbaEngine_Java_API_Reference.

Simba.NET

Simba.NET provides a complete ADO.NET interface and all the processing required to meet the ADO.NET specification. It is the connection between your custom DSI implementation and common ADO.NET reporting applications such as Microsoft SQL Server Analysis Services.

The Data Store Interface (DSI)

The data store interface, or DSI, defines a generic view of an SQL database that is independent of the industry standards for data access, such as ODBC, JDBC, ADO.NET and OLE DB. The Simba SDK translates the ODBC, JDBC, ADO.NET and OLE DB interfaces to the DSI in C++, Java, or C#. By writing code to map from the DSI to your data store, you are creating a connector that can use one of these standard interfaces.

The DSI API is object-oriented and simpler to use than the industry standard interfaces. This makes it easier to write a DSI implementation that will translate to your custom data store. Also, because the DSI API provides a consistent API whether you are implementing ODBC, JDBC, ADO.NET or OLE DB, it is easier to re-use your knowledge when creating a connector for a different industry standard.

The Data Store Interface (DSI)

The data store interface, or DSI, defines a generic view of an SQL database that is independent of the industry standards for data access, such as ODBC, JDBC, ADO.NET and OLE DB. The Simba SDK translates the ODBC, JDBC, ADO.NET and OLE DB interfaces to the DSI in C++, Java, or C#. By writing code to map from the DSI to your data store, you are creating a connector that can use one of these standard interfaces.

Simba Client/Server

Simba Client/Server allows remote access to your data store. You link the SimbaServer component with your DSII to create a custom connector than can accept requests from SimbaClient. You deploy SimbaClient with the application to send requests to the remote connector.

SimbaServer is most frequently used as a stand-alone executable, although it can be set up as a DLL or shared object under another server.

The DSI implementation used with SimbaServer can choose to include the Simba SQLEngine. It can be written to perform a wide range of functionality including SQL query processing with Simba SQLEngine, concentrating client requests through one executable, aggregating data stores, or controlling data access through role-based permissions. There are many possibilities for using SimbaServer as an intermediate processing step in a larger system.

The SimbaServer can be written in C++, or written in Java including the JNI Server. For more information see the *SimbaClient/Server Developer Guide*.

SimbaClient for ODBC

SimbaClient for ODBC is an ODBC connector DLL or shared object that can connect to SimbaServer. It includes SimbaODBC and a DSI implementation that communicates via the Simba Client/Server protocol to SimbaServer. Since any SQLEngine in the stack will be on the server side, there is no need for Simba SQLEngine in this connector. This is a completely generic ODBC connector that, when queried, reports the capabilities of the database that is connected to SimbaServer.

Note:

SimbaClient for ODBC is provided by Simba, and is ready to deploy with no additional development effort required.

SimbaClient for JDBC

SimbaClient for JDBC is a JDBC connector packaged as a .jar file so you can install it in an end user's client-side Java Run Time Environment. SimbaClient for JDBC includes the equivalent of SimbaODBC and custom Java code that communicates via the Simba Client/Server protocol with SimbaServer.

Note:

SimbaClient for JDBC is provided by Simba, and is ready to deploy with no additional development effort required.

C++ to Java Bridge (JNI DSI)

This component of the Simba SDK allows you to write the DSII in Java, then link to SimbaODBC or SimbaServer (including Simba SQLEngine) to create an ODBC connector.

C++ to C# Bridge (CLI DSI)

This component of the Simba SDK allows you to write the DSII in C#, then link to SimbaODBC or SimbaServer (including Simba SQLEngine) to create an ODBC connector.

Sample Connectors

The Simba SDK includes a number of sample connectors and sample connector projects to help you get started quickly with your custom ODBC or JDBC connector. For more information on sample connectors, see "Sample Connectors and Projects" below.

Related Topics

Simba SDK C++ API Reference

Simba SDK Java API Reference

Simba SDK FAQ

Sample Connectors and Projects

The Simba SDK includes a number of sample connectors and sample connector projects to help you get started quickly with your custom ODBC or JDBC connector. The compiled C++ sample connectors are in the `Examples\Builds\Bin` folder of your Simba SDK installation directory, the compiled Java sample connectors are in `Examples\Builds\Lib`, and the sample connector projects are in `Examples\Source`.

Note:

This guide explains how to build a connector for data stores that support SQL. If you want to build a connector for data stores that do not support SQL, see Developing Connectors for Data Stores Without SQL.

You may find the HTML version of this guide easier to use. See Developing Connectors for SQL-capable Data Stores.

Getting Started with the Sample Connector Projects

The sample connector projects are a great way to get started developing your custom connector. Each sample connector is accompanied by a 5-Day Guide, which walks you through the steps of building, configuring, and customizing the project.

For information on how to use the sample connector projects, see 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

The following sections describe each of the sample connector projects and sample connectors.

Note:

Although this guide explains how to build a connector that does not use the SQL Engine, information on using the SQL Engine is included in this section for reference.

Quickstart Sample Connector

C++	Not SQL-Capable	ODBC

Quickstart is a C++ sample DSI implementation of an ODBC connector that reads text files in tabbed Unicode text format. This is not a SQL aware data source, so the Simba SQLEngine component is employed to perform the necessary SQL processing. This sample's purpose is to provide a simple, working connector that you can copy and transform into a connector that accesses your non-SQL data store. An ODBC configuration DLL is included.

The 5-Day Guide for the Quickstart sample connector project is located at 5 Day Guides at http://www.simba.com/resources/sdk/documentation/ under the section "Build a C++ ODBC Connector in 5 Days". Select one of the guides for a non-SQL-based data source.

DotNetQuickstart Sample Connector

C#	Not SQL-Capable	ODBC

DotNetQuickstart is a C# sample DSI implementation that is the same as the Quickstart sample above, except that it is written in C# using Simba's C++ to C# bridge API (also referred to as the CLIDSI API). See the document, "Build a C# ODBC Connector in 5 Days" for a step-by-step walk-through of the process of creating a custom ODBC connector using C#.

JavaQuickstart Sample Connector

Java	Not SQL-Capable	ODBC

JavaQuickstart is a Java sample DSI implementation that is the same as the Quickstart sample, except that it is written in Java using Simba's C++ to Java bridge API (also referred to as the JNIDSI API). See the document "Build a Java ODBC Connector in 5 Days" for a step-by-step walk-through of the process of creating a custom ODBC connector using Java.

Ultralight Sample Connector

C++	SQL-Capable	ODBC

Ultralight is a sample connector that illustrates how to build a DSII for a database that already supports SQL and therefore does not require the SQLEngine component.

The Ultralight example does not truly support SQL; rather, it simply looks for keywords in the query and returns a hardcoded result set. Nevertheless, this is sufficient to show all the necessary building blocks and provide a placeholder where your real SQL processing and result set generation could take place.

The 5-Day Guide for the Ultralight sample connector project is located at 5 Day Guides at http://www.simba.com/resources/sdk/documentation/ under the section "Build a C++ ODBC Connector in 5 Days". Select one of the guides for a SQL-based data source.

DotNetUltralight Sample Connector

C#	SQL-Capable	ODBC

DotNetUltralight is a C# sample DSI implementation that is the same as the Ultralight sample above, except that it is written in C#. DotNetUltralight can be built using either Simba.ADO.NET or using Simba's C++ to C# bridge API (also referred to as the CLIDSI API). When using Simba.ADO.NET, the resulting connector will be written entirely in C#, providing an ADO.NET interface. When using Simba's C++ to C# bridge API, the resulting connector will be a mixture of C# and C++, providing an ODBC interface or SimbaServer executable for use with any of the SimbaClient connectors.

JavaUltralight Sample Connector

Java	SQL-Capable	JDBC	ODBC

JavaUltralight is a Java sample DSI implementation that is the same as the Ultralight sample above, except that it is written in Java. JavaUltralight can be built using either SimbaJDBC or using Simba's C++ to Java bridge API (also referred to as the JNIDSI API). When using SimbaJDBC, the resulting connector will be written entirely in Java, providing a JDBC 4.0, 4.1, or 4.2 interface. When using Simba's C++ to Java bridge API, the resulting connector will be a mixture of Java and C++, providing an ODBC interface or SimbaServer executable for use with any of the SimbaClient connectors.

The 5-Day Guide for the JavaUltralight sample connector project is located at 5 Day Guides at http://www.simba.com/resources/sdk/documentation/ under the section "Build a JDBC Connector in 5 Days". Select the guide for a SQL-based data source.

JavaQuickJson Sample Connector

Java	Not SQL-Capable	JDBC	ODBC

JavaQuickJson is a sample Java DSI imlplementation written purely in Java to demonstrate usage of the Java Simba SQLEngine. The connector accesses a data store comprised of JSON files and uses a third party JSON API to read and write data to those files. By including the Java Simba SQLEngine, the connector demonstrates how to implement some of the classes specific to the Java Simba SQLEngine and how the Java Simba SQLEngine can be used to access a data store that is not organized using traditional tables and columns.

The 5-Day Guide for the JavaQuickJson sample connector project is located at 5 Day Guides at http://www.simba.com/resources/sdk/documentation/ under the section "Build a JDBC Connector in 5 Days". Select the guide for a non-SQL-based data source.

Using the Sample Connectors for Debugging

You can use the sample connectors to analyze and debug suspected problems in your custom connector. For example, if you think your DSI implementation is working correctly but there is a problem in your Simba SDK system, there is a simple way to determine where the problem lies. If the problem

shows up when you run the system you have assembled with the sample connector project implementation, then the problem is likely to be in Simba SDK components.

If the problem goes away when you replace your DSI implementation with the sample connector project, then you need to do some more investigation of your implementation. In either case, analysis and debugging is focused and reduced, lowering your cost to deliver a solution to your customers

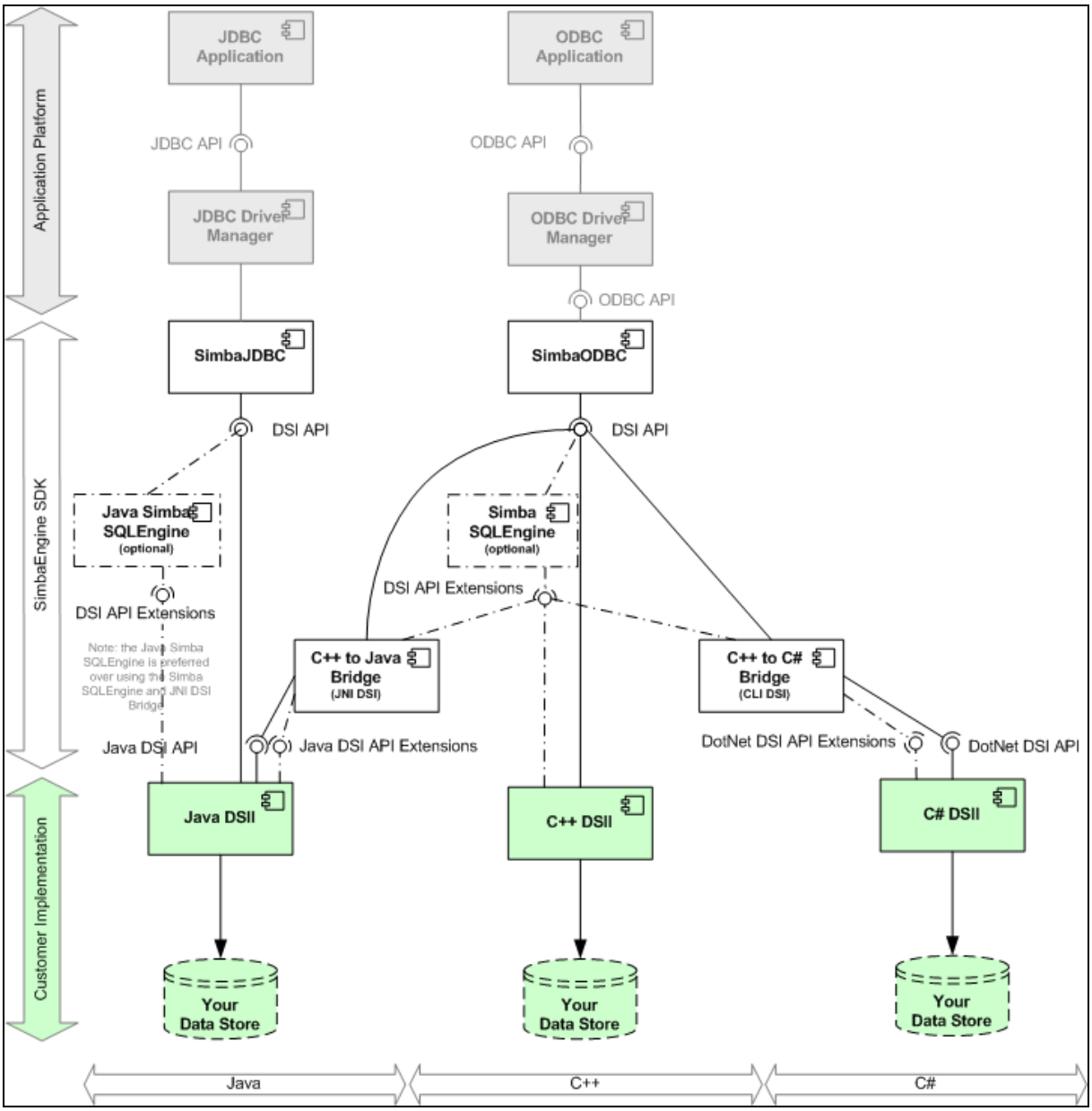
Related Topics

Building Blocks for a DSI Implementation

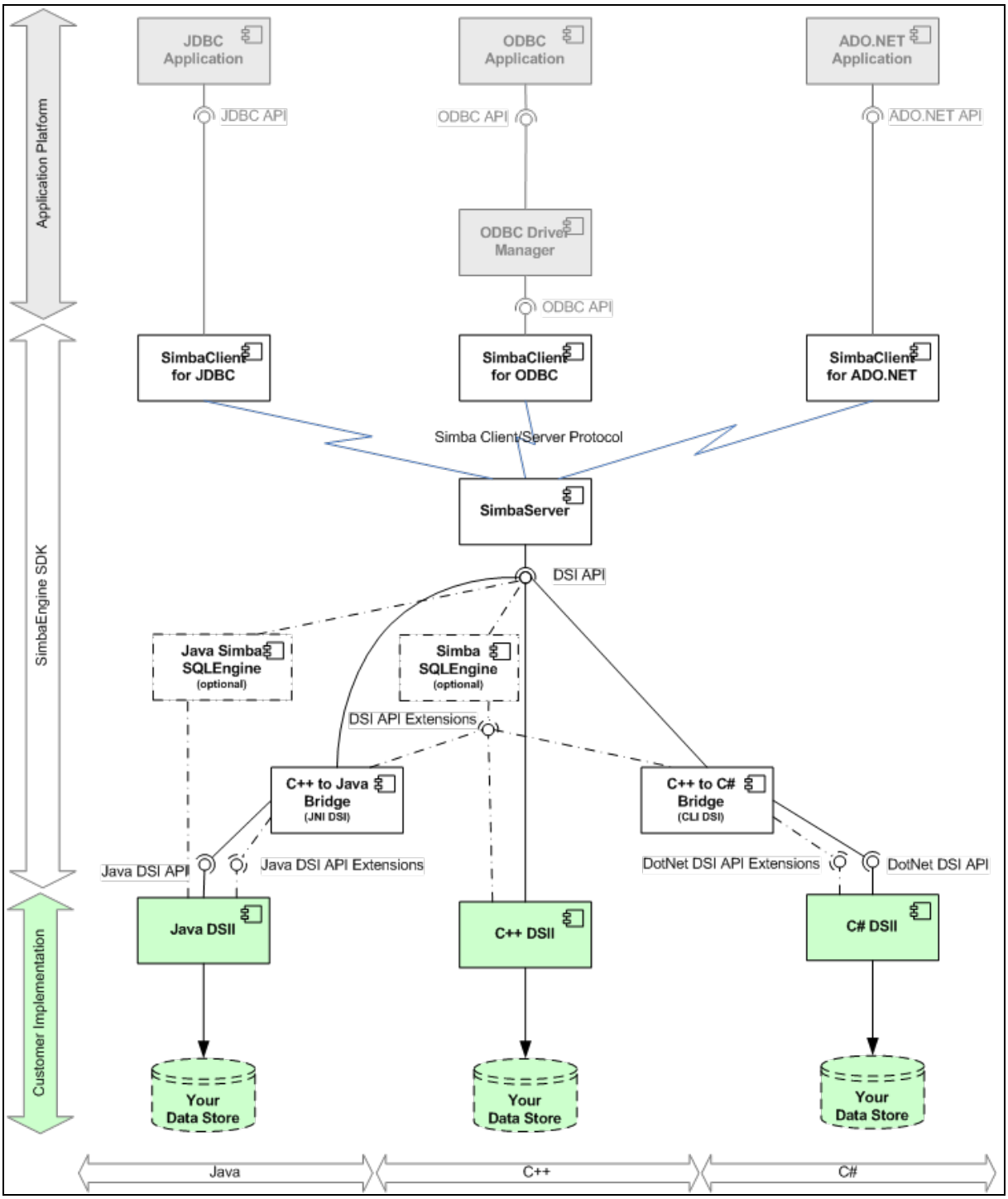
The diagrams in this section illustrate the how the Simba SDK components work together in both the standalone and the client/server deployment.

Note:

Although this guide explains how to build a connector that does not use the SQL Engine, information on using the SQL Engine is included in this section for reference.



Standalone Deployment



Client/Server Deployment

Each of these diagrams has three zones horizontally and vertically. The horizontal zones are:

Zone	Description
Application Platform	These elements, shown in gray boxes, represent the client-side applications that will connect to the completed ODBC, JDBC connector, or ADO.NET provider that you build with the SDK.
Simba SDK	These elements, shown in white boxes, are the components that make up the SDK itself.
Customer Implementation	These elements, shown in green boxes, represent the unique code you write to access your data store.

The vertical zones align with the different development environments available to you:

Zone	Description
C++	A C++ DSII may be written to support ODBC applications by linking upward from your implementation to the SimbaODBC component. Alternately, you can also support JDBC or ADO.NET applications by linking upward to the SimbaServer component.
Java	A Java DSII may be written to support JDBC applications by linking to the SimbaJDBC component. Alternately, you can support ODBC applications by linking upward through the C++ to Java Bridge to the SimbaODBC component, or support ODBC or ADO.NET applications by linking your Java DSII upward via the same bridge to the SimbaServer component.
C#	A C# DSII may be written to support ADO.NET applications by linking to the Simba.NET component. Alternately, you can support ODBC applications by linking upward via the C++ to C# Bridge to the SimbaODBC component, or support ODBC or JDBC applications by linking your C# DSII upward via the same bridge to the SimbaServer component.

Related Topics

Simba SDK FAQ

Getting Started

To get started building a custom ODBC, JDBC, OLE DB, or ADO.net connector using the Simba SDK, follow these general steps:

1. Plan how to map your data store schema to the DSI model.
2. Use one of the 5-Day Guides to set up your development environment. For more information on the 5-Day Guides, see http://www.simba.com/drivers/simba-engine-sdk/#documentation.
3. Implement your plan for translating your data store to the DSI.

Mapping Your Data Store Schema to the DSI Model

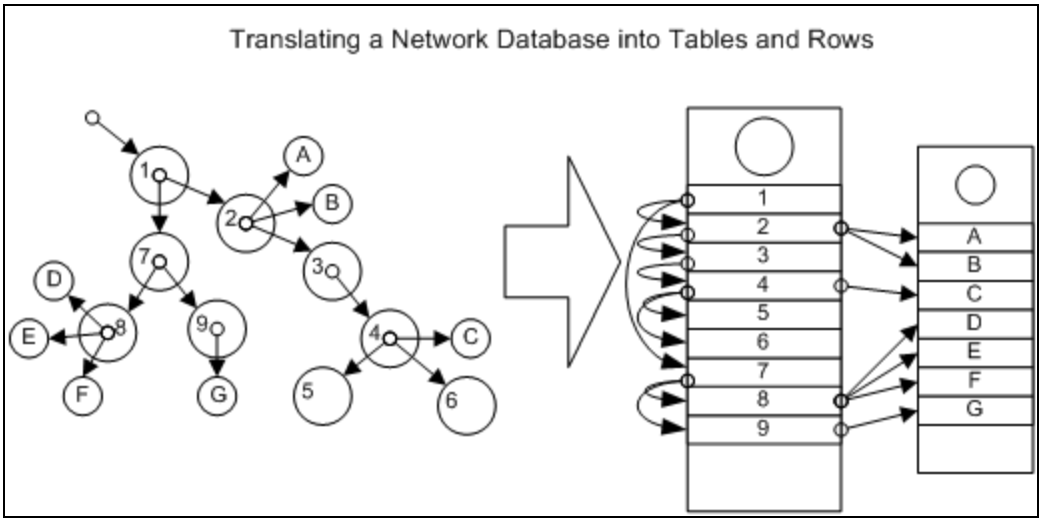
The first step is to map your data store to the DSI model. The DSI represents the data store as a series of tables and columns, and the purpose of your DSI implementation is to translate your real data store schema into the DSI representation.

Note:

The DSI represents the data store as a series of tables and columns. The purpose of your DSI implementation is to translate your data store schema into tables and columns that the DSI can understand.

The diagram below shows an example of one type of mapping. The database on the left uses an object-oriented or networked schema to store the data. However, a relational SQL execution engine cannot directly use this schema.

If you represent the same database as tables and columns, even though you do not actually transform the database into this new form, it fits the relational paradigm. Now you can write a DSI implementation to create this view of the data and Simba SDK can use the SQL Engine to execute SQL queries against it. In this way, any database you can represent as tables and columns can be accessed by Simba SDK and made accessible to applications and reporting tools.



Relational applications, or applications that can access relational databases, cannot access the networked data store on the left, because it is not a relational database. Once you translate the data store to use tables and columns, as shown in the data store on the right, relational applications can use the SQL Engine to send SQL queries to it. Simba SQLEngine works with tabular database schema (but not non-tabular database schema).

Using Virtual Tables

It might be tempting to create a tabular view of your data store by reading the entire database into temporary tables, and accessing these tables through the DSI. However, this is inefficient and only works for very small databases. A more efficient method is to create virtual tables and then access the original database when the Simba SDK requests data through the DSI.

Related Topics

Build a Connector in 5 Days

Frequently Asked Questions

This section answers the questions that are commonly asked by people who are new to the Simba SDK product and technology. For a more detailed FAQ, see the *Testing and Troubleshooting* section of this guide.

What Platforms does the Simba SDK Support?

For information about the supported versions of Windows, Unix, Linux, and macOS, plus a list of supported compilers, see "Supported Platforms" on page 117.

What is ODBC?

ODBC stands for Open Database Connectivity (ODBC). It is a C-language open standard Application Programming Interface (API) for accessing relational databases.

In 1992, Microsoft contracted Simba to build the world's first ODBC connector; SIMBA.DLL, and standards-based data access was born. Using ODBC, you can access data stored in many common databases. A separate ODBC connector is needed for each database to be accessed. An ODBC Driver Manager is also needed. This is supplied with the Windows operating system, and is available commercially and as open source on Unix and Linux.

What is MDAC?

MDAC, or Microsoft Data Access Components, are runtime components that are shipped with the Windows operating system. These components contain interfaces for ODBC, OLEDB and ADO, as well as the ODBC connectors for Microsoft's database-related products.

The MDAC SDK is available from the Microsoft Developer Network (MSDN) and can be downloaded from: http://www.microsoft.com/downloads/en/details.aspx?familyid=5067FAF8-0DB4-429A-B502-DE4329C8C850&displaylang=en.

In newer versions of Windows (Vista & 7), MDAC is now called Windows DAC. For more information, see http://msdn.microsoft.com/en-us/library/ms692877%28v=vs.85%29.aspx.

What Third-Party Components Does the Simba SDK Use?

For information on the third-party components used by the Simba SDK, see "Third Party Licenses" on page 183.

I am new to ODBC. How does my application work with an ODBC Connector?

ODBC-enabled applications always access ODBC connectors through the Driver Manager that is installed on the operating system. An instance of the Driver Manager is created for each ODBC application. The application will specify to the Driver Manager which ODBC connector to use when establishing a connection. The Driver Manager will then load the appropriate ODBC connector. Once

the ODBC connector is loaded, the Driver Manager will map all incoming requests to the appropriate functions exported by the ODBC connector.

To interact with a Driver Manager, ODBC-enabled applications will request the following three ODBC handles:

SQL_HANDLE_ENV

Represents an environment handle. Every instance of an ODBC connector will be associated with a single environment handle.

SQL_HANDLE_DBC

Represents a connection handle. Connections are created using one of the following three ODBC methods: SQLConnect(), SQLBrowseConnect(), SQLDriverConnect(). Every connection handle is associated with its parent environment handle.

SQL_HANDLE_STMT

Represents a statement handle. Every statement that is to be executed via ODBC will be associated with its own statement handle. Every statement handle is associated with its parent connection handle.

The Driver Manager interacts with an ODBC connector in much the same way. The Driver Manager will request the handles for the environment, connection and statement. All calls made from the ODBC-enabled application to the Driver Manager require the Driver Manager allocated handle and will be implemented as follows:

1. Map incoming Driver Manager allocated handle to an instance representing the handle.
2. Call the ODBC connector associated with the instance using the ODBC connector associated handle.

What is ICU?

ICU stands for the International Components for Unicode (ICU) libraries. These libraries provide Unicode handling mechanisms on which the SimbaODBC components are dependent. These libraries are distributed under an open source license at:

http://source.icu-project.org/repos/icu/icu/trunk/license.html

ICU is freely available from:

http://www.icu-project.org/download

What is SimbaODBC?

SimbaODBC is a component part of Simba SDK for developing full-featured, optimized ODBC 3.80 connectors on top of any SQL-enabled data source. SimbaODBC provides extensibility for JDBC, OLE DB as well as ADO.NET connectivity. SimbaODBC simplifies exposing the query parsing, query execution and data retrieval facilities of your SQL-enabled data source.

What do the Different Components of SimbaODBC do?

SimbaODBC ships with a number of static libraries. You will link these libraries into the code you write to communicate with an underlying SQL-92 enabled data store.

What SQL Conformance Level Does Simba SDK support?

ODBC specifies three levels of SQL grammar conformance: Minimum, Core and Extended. Each higher level provides more fully-implemented data definition and data manipulation language support. Simba SDK fully supports Core DML SQL grammar, as well as many Extended grammars.

Related Topics

5 Day Guides at http://www.simba.com/resources/sdk/documentation/

Core Features

This section explains the features that most custom connectors will implement.

Fetching Metadata for Catalog Functions

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

Your custom ODBC connector uses metadata sources, provided by the Simba SDK, to handle SQL catalog functions. Of the 13 `DSIMetadataSource` sub-classes, there is only one that you need to modify to make a basic connector work. This section describes the other metadata classes and under what circumstances you need to update them.

Implementation

Your `CustomerDSIIDataEngine` class has to derive from `IDataEngine` or `DSIDataEngine`. If it is derived from `IDataEngine`, then the following function has to be implemented:

Simba::DSI::IResult* MakeNewMetadataResult(

Simba::DSI::DSIMetadataTableID in_metadataTableID,

const std::vector<Variant>& in_filterValues,

const simba_wstring& in_escapeChar,

const simba_wstring& in_identifierQuoteChar,

bool in_filterAsIdentifier);

This function creates a new `IResult*` which contains a metadata data source and filters out rows in the metadata table that are not needed. If the connector does not support a metadata table, then the metadata source in the `IResult*` should be an empty metadata data source with no rows.

The function takes the following parameters:

- in_metadataTableID: Identifier to create the appropriate metadata table. For a list of the possible identifiers, refer to the table below. For complete details on each identifier, refer to DSIMetadataTableID.h in the API guide.

- in_filterValues: Filters to be applied to the metadata table. These filters are passed in by the application that calls the catalog function and cannot be modified. For example, the catalog function `SQLTables` contains the arguments `CatalogName`, `SchemaName`, `TableName`, and `TableType`. These arguments are extracted to the `in_filterValues` vector.
  While these values cannot be modified, if the CatalogName is NULL, the current catalog name is used.

- in_escapeChar: Escape character used in filtering.

- in_identifierQuoteChar: Quote identifier, which is the quotation mark that this filter recognizes.

- in_filterAsIdentifier: Indicates if string filters are treated as identifiers. This can be set through the connection attribute SQL_ATTR_METADATA_ID.

If it is derived from `DSIDataEngine`, then the following function has to be implemented:

Simba::DSI::DSIMetadataSource* MakeNewMetadataTable(

Simba::DSI::DSIMetadataTableID in_metadataTableID,

Simba::DSI::DSIMetadataRestrictions& in_restrictions,

const std::vector<Simba::Support::Variant>& in_filterValues,

const simba_wstring& in_escapeChar,

const simba_wstring& in_identifierQuoteChar,

bool in_filterAsIdentifier);

This function creates a new `Metadatasource*` which contains raw metadata. If the connector does not support a metadata table, then it should return an empty metadata source with no rows by returning a `DSIEmptyMetadataSource` object.

The function takes the following parameters:

- `in_metadataTableID`: Identifier to create the appropriate metadata table. For a list of the possible identifiers refer to the table below. For complete details on each identifier refer to DSIMetadataTableID.h in the API guide.

- `in_restrictions`: Restrictions that may be applied to the metadata table. Map of DSIOutputMetadataColumnTag that identify columns in the result set, to the restriction that apply to those columns. For example, if the DSIOutputMetadataColumnTag identifies a catalog name, then the restriction specifies that the result set should only contain rows with the same catalog name as the restriction. For a complete list and details of DSIOutputMetadataColumnTag values, refer to DSIMetadataColumnIdentifierDefns.h in the API guide.

- `in_filterValues`: Filters to be applied to the metadata table. These filters are passed in by the application that calls the catalog function and cannot be modified. For example, the catalog function `SQLTables` contains the arguments `CatalogName`, `SchemaName`, `TableName`, and `TableType`. These arguments are extracted to the `in_filterValues` vector.
  While these values cannot be modified, if the CatalogName is NULL, the current catalog name is used.

- `in_escapeChar`: Escape character used in filtering.

- `in_identifierQuoteChar`: Quote identifier, which is the quotation mark that this filter recognizes.

- `in_filterAsIdentifier`: Indicates if string filters are treated as identifiers. This can be set through the connection attribute SQL_ATTR_METADATA_ID.

If the metadata table is supported by the connector, then a new class should be implemented by deriving from `Simba::DSI::DSIMetadataSource` and implementing all the functions.

Note:

The Ultralight connector is a sample connector derives `ULDataEngine` from `DSIDataEngine`. It implements classes for metadata tables for DSI_TABLES_METADATA, DSI_CATALOGONLY_METADATA, DSI_SCHEMAONLY_METADATA, DSI_TABLETYPEONLY_METADATA, DSI_COLUMNS_METADATA, and DSI_TYPE_INFO_METADATA metadata table identifiers.

Adding Custom Metadata Columns

Each catalog function returns data as a result set. In addition to the ODBC-standard columns that are returned when a catalog function is executed, the data store can return additional columns. Your custom connector can add custom metadata columns to the Metadata result tables in order to support data source-specific data. The `DSIMetadataSource`-derived classes support custom columns, which are enabled by proper implementations of several functions. These functions are:

- `GetCustomColumns`
- `GetCustomMetadata`

Note:

- All custom metadata columns must be of type `DSICustomMetadataColumn`. The header file for `DSICustomerMetadataColumn` can be found at `[INSTALL_DIRECTORY]\DataAccessComponents\Include\DSI\Client\DSICustomMetadataColumn.h`
- This feature is only supported in the C++ SDK.

A sample implementation of a custom metadata column for `CustomerDSIITablesMetadataSource` is shown below. Adding custom metadata columns to any other metadata source follows a similar formula.

To Add Custom Metadata Columns:

1. Define a custom column tag for the custom column:

   ```
   const simba_uint16 CUSTOM_TABLES_COLUMN_TAG = 50;
   ```

2. Define a member variable for the custom column:

   ```
   std::vector<Simba::DataSToreInterface::DataEngine::Client::
    DSICustomMetadataColumn*> m_customMetadataColumns;
   ```

3. Initialize the metadata for the custom columns in the `CustomerDSIITablesMetadataSource` constructor. Use the static `MakeNewSqlTypeMetadata` function of the `Simba::Support::TypedDataWrapper::SqlTypeMetadataFactory` class.

```
using namespace Simba::DSI;

using namespace Simba::Support;

DSICustomMetadataColumn* column = NULL;

DSIColumnMetadata* colMetadata = NULL;

SqlTypeMetadata* metadata = NULL;

// Custom column
```

```
colMetadata = new DSIColumnMetadata();
colMetadata->m_autoUnique = false;
colMetadata->m_caseSensitive = false;
colMetadata->m_label = L"CUSTOM_COL";
colMetadata->m_name = L"CUSTOM_COL";
colMetadata->m_unnamed = false;
colMetadata->m_charOrBinarySize = 128;
colMetadata->m_nullable = DSI_NULLABLE;
colMetadata->m_searchable = DSI_PRED_NONE;
colMetadata->m_updatable = DSI_READ_ONLY;
// Create SqlTypeMetadata*
metadata = SqlTypeMetadataFactorySingleton::GetInstance()>CreateNewSqlTypeMetadata(SQL_
VARCHAR);
column = new DSICustomMetadataColumn(
metadata,
colMetadata,
CUSTOM_TABLES_COLUMN_TAG);
m_customColumnMetadata.push_back(column);
```

4. For information on `DSIColumnMetadata`, refer to Simba SDK Java API Reference or Simba SDK C++ API Reference.

5. Implement `CustomerDSIITablesMetadataSource::GetCustomColumns`:

```
void CustomerDSIMetadataSource::GetCustomColumns
(std::vector<Simba::DSI::DSICustomMetadataColumn*>& out_customColumns)
```

6. Iterate over `m_customColumns` and push them into `out_customColumns`.

7. Implement `CustomerDSIITablesMetadataSource::GetCustomMetadata`:

   ```
   bool CustomerDSIITablesMetadataSource:::GetCustomMetadata(
   simba_uint16 in_columnTag,
   SqlData* in_data,
   simba_signed_native in_offset,
   simba_signed_native in_maxSize)
   ```

   The implementation is the same as `CustomerDSIITablesMetadataSource::GetMetadata` except the column tags you check are your custom column tags.For example:

   ```
   switch (in_columnTag){
   ```

```
        case CUSTOM_TABLES_COLUMN_TAG:{
        //retrieve the appropriate data from your m_result
        }
        default:{
        //throw exception – metadata column not found.
        }
}
```

Overriding the Value of Default Properties

ODBC and JDBC connectors use connection, connector, environment, and statement properties to specify and define their behavior and capabilities. The Simba SDK provides default values for these properties. If the capabilities of your custom connector are different from the specified defaults, or if you need to support the requirements of a specific application, you can override these default values.

The Simba SDK implements these properties in the following classes in the Core library:

Property Type	Class	Name of Property Map
Connection properties	`DSIConnection`	`m_connectionProperties`
Connector properties	`DSIDriver`	`m_driverProperties`
Environment properties	`DSIEnvironment`	`m_environmentProperties`
Statement properties	`DSIStatement`	`m_statementProperties`

Properties are represented as key-value string pairs, which are stored in a property map as shown in the table above. Properties are initialized with their default value in the constructor of the corresponding class.

You can override these properties in your DSII subclass of the corresponding Core class, but you must only override the default value for any property during the construction of each class instance. After that, property changes should only come from the ODBC application calling the appropriate API function. The one exception to this rule is that connection properties may be updated at the time a connection is successfully established. This should be done before returning from the `CustomerDSIIConnection::connect` function.

Each of these four Core classes has a function called `SetProperty`, which is used to set the value for a property or attribute.

For a description of properties and default values in the C++ SDK, see the Simba SDK C++ API Reference. Select **Namespaces** -> **Simba::DSI** then see the following enumerations:

- `DSIConnPropertyKey`
- `DSIDriverPropertyKey`
- `DSIEnvPropertyKey`
- `DSIStmtPropertyKey`

For a description of properties and default values in the Java SDK, see the following classes in the Simba SDK Java API Reference:

- `ConnPropertyKey`
- `DriverPropertyKey`
- `EnvPropertyKey`
- `StmtPropertyKey`

Note:

The term "property" and "attribute" are used interchangeably in the Simba SDK. For example, a method might be called `GetProperty` but work with `AttributeData` objects.

Example: Overriding the Value of Connection Properties

The example in this section shows how to override default property and attribute values for the `DSIConnection` class. You can use the same method to override default values in the `DSIStatement`, `DSIDriver` and `DSIEnvironment` classes.

Note:

This example is in C++ but it also applies to the Java SDK.

The example subclass of `DSIConnection` is called `CustomerDSIIConnection`. In the `CustomerDSIIConnection` constructor, use the `DSIConnection::SetProperty()` method to set the property or attribute value. The signature of the `DSIConnection::SetProperty` function is:

```
virtual void SetProperty(
    DSIProperties::DSIConnPropertyKeys::DSIConnPropertyKey in_key,
    Simba::Support::Utility::AttributeData* in_value)
```

Example: Set the server name

The default value for DSI_SERVER_NAME is “”. It should be set to the name of the DSI server. Pass in the key for the server name and the name of the server to the `SetProperty` function.

```
SetProperty(
    DSIProperties::DSIConnPropertyKeys::DSI_SERVER_NAME,
    Utility::AttributeData::MakeNewWStringAttributeData(<name_of_server>)
```

);

Example: Specify the Supported SQL_CHAR Conversions

The default value for DSI_SUPPORTED_SQL_CHAR_CONVERSIONS is DSI_CVT_CHAR. If the application supports more conversions, you need to change this value. Here, the value for the DSI_SUPPORTED_CHAR_CONVERSIONS property is made up of a concatenation of all the values provided.

```
SetProperty(
    DSIProperties::DSIConnPropertyKeys::DSI_SUPPORTED_SQL_CHAR_
    CONVERSIONS,
    Utility::AttributeData::MakeNewUInt32AttributeData(
    DSIProperties::DSIConnPropertyValues::DSI_CVT_CHAR |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_NUMERIC |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_DECIMAL |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_INTEGER |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_SMALLINT |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_FLOAT |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_REAL |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_VARCHAR |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_LONGVARCHAR |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_BINARY |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_VARBINARY |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_BIT |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_TINYINT |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_BIGINT |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_TIMESTAMP |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_LONGVARBINARY |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_WCHAR |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_WLONGVARCHAR |
    DSIProperties::DSIConnPropertyValues::DSI_CVT_WVARCHAR)
);
```

Implementing Logging

The Simba SDK includes comprehensive logging functionality that you can use when developing and troubleshooting your connector.

For information on how to turn on logging in the sample connectors, see *Enable Logging* in the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

For information on logging to Event Tracing for Windows (ETW), see "Logging to Event Tracing for Windows (ETW)" on page 161.

The Simba SDK enables multiple logger objects logging to separate files: one for the single `IDriver` instance, and one for each `IConnection` instance. This allows for easier debugging of threading issues, while still allowing for logging of issues that happen before a connection is established. If only one central log is needed, then child `IConnection` objects can return the parent `IDriver` log instance to have all logging calls focus on one `ILogger`.

The `ILogger` has a default implementation in `DSILog`, each of which logs to a file. There are several functions to log messages at varying levels of importance as needed. The `DSILog` allows for filtering of logging messages based on both log level and namespace, enabling you to narrow logging to suspect areas of your DSII. If the default `DSILog` does not provide enough functionality, then you may choose to create a full implementation of `ILogger` directly from the interface that provides the functionality that you need.

Log Settings

There are three settings that affect logging by default:

- **LogLevel** – Used to set the level of logging that is performed. Valid values are:
  - 0 or “Off”
  - 1 or “Fatal”
  - 2 or “Error”
  - 3 or “Warning”
  - 4 or “Info”
  - 5 or “Debug”
  - 6 or “Trace”
- **LogPath** – Set the path that the default logging implementation will create the log files in. Defaults to the current working directory.
- **LogNamespace** – Filters the logging based on the namespace/package that the messages are coming from. For instance, the value “Simba” will filter all logging messages to namespaces starting with “Simba” such as “Simba::Support”.

The settings are read from the registry at **HKLM\SOFTWARE\<OEM NAME>\Driver** for both SimbaODBC and Simba.NET, while they are read from the connection string for SimbaJDBC.

For Simba.NET on platforms using .NET Core, there may be no registry to read configuration from if not using Windows. Instead, the configuration can be read from one of several configuration files:

1. User-level configuration for the current application: `%APPDATA%/[COMPANY]/[APPLICATION]/[APPLICATION VERSION]/user.config`.
   (`%APPDATA%` is typically `C:\Users\username\AppData\Roaming` on Windows and `/home/username/.config/` on other operating systems.)
2. User-level configuration for the provider: `%APPDATA%/[BRANDING]/[BRANDING].config`.

3. Application-level configuration: *[APPLICATION DIRECTORY]/[APPLICATION NAME]*.config.
4. Provider-level configuration: *[PROVIDER DIRECTORY]/[BRANDING]*.config.

The format of the configuration file is the same as a typical .NET App.Config file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<configSections>
<section name="Simba.UltraLight" type="Simba.DotNetDSI.ConfigReader,
Simba.DotNetDSI" />
</configSections>
<Simba.UltraLight
LogLevel="0"
LogPath="/tmp/ultralight.log" />
</configuration>
```

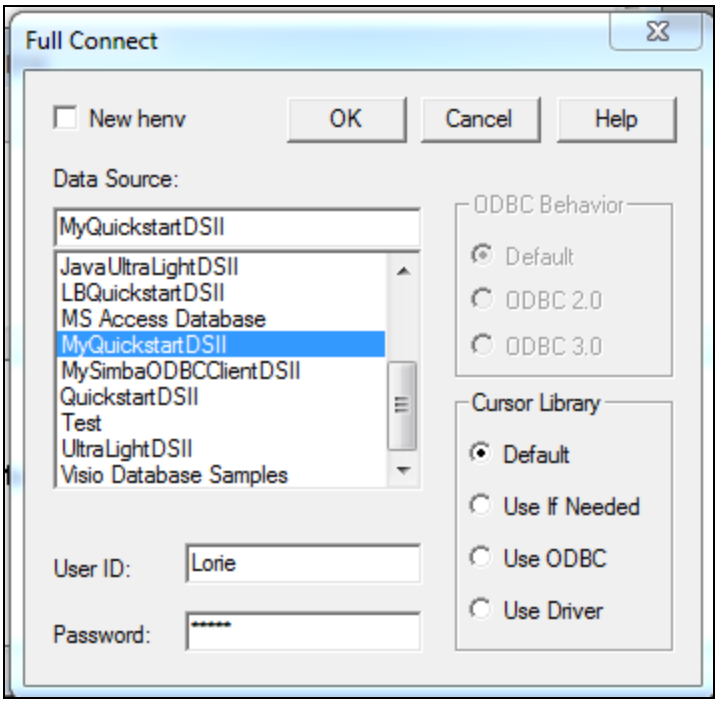
The section name and name of the tag added are based on the branding set in the provider, with "\" replaced by " . ". If adding to an existing configuration file, the <section> tag should be added to an existing <configSections> tag.

Hiding Sensitive Information in the Log Files

The Simba SDK does not log the value of the connection parameters username (UID) and password (PWD). Instead, the values are logged as asterisks (****). The DSIConnection::IsSensitiveAttribute() method determines whether or not the value of a connection parameter should be logged. The IsSensitiveAttribute() method is called by the ConnectionSettings class when a connection is established.

Example

If you enter a username and password when connecting to the MyQuickstartDSII connector, the resulting log file will contain the strings "PWD" = "***" and "UID" = "***", rather than the actual username and password.



Your custom ODBC connector can specify additional connection parameters that should not have their values logged in plain text. To do this, override `DSIConnection::IsSensitiveAttribute()` in your `Connection.cpp` class.

For example, in the following code, the values of the connection parameters `Secret1` and `Secret2` will be logged as asterisks (`*****`):

```
bool QSConnection::IsSensitiveAttribute(const simba_wstring& in_attribute)
{
    if ((in_attribute.IsEqual("Secret1")) ||(in_attribute.IsEqual("Secret2")))
        {
            return true;
        }
    return DSIConnection::IsSensitiveAttribute(in_attribute);
}
```

Logging in the Java DSI

The Java Simba SDK includes a helper class called `LogUtilities` to help you implement logging functionality. This class provides a copy of many of the functions that exist in `ILogger`, but the functions take an `ILogger` instance and do not take the namespace or class names from which the logging call originates. Instead, it uses reflection to determine that information, easing use of the logger.

Logging in the DotNet DSI

The dotNet Simba SDKincludes a helper class called `LogUtilities` to help you implement logging functionality. This class provides a copy of many of the functions that exist in `ILogger`, but the functions take an `ILogger` instance and do not take the namespace or class names from which the logging call originates. Instead, it uses reflection to determine that information, easing use of the logger.

Simba.NET Specific Features

Note that there is an extra setting for Simba.NET to provide logging if an error occurs before a DSI DLL is loaded:

- PreloadLogging – Set to 0 (off) or 1 (on) to log to the file InitialDotNet.log. Once a DSII DLL is loaded, the DSI ILogger will be used.

Related Topics

"Logging to Event Tracing for Windows (ETW)" on page 161

Enable Logging in the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

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

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

Adding Custom Connection and Statement Properties

Custom properties can be added to `Connection` and `Statement` objects. These properties allow you to customize how your connection and statement objects behave.

> **Important:**
> **Important:**
>
> Before you can implement custom properties for your connection and statement attributes, you should request and reserve a value for each attribute from the Open Group. This ensures that no two connectors will assign the same integer value to different custom attributes. If you do not reserve a unique attribute or use one that is already in use, your connector may experience compatibility issues with any application that uses the conflicting custom attributes for other connectors.
>
> For more information on requesting a value from the Open Group, refer to the *Connector-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes* section of the MSDN ODBC Programmer's Reference.

Custom Properties in the C++ SDK

You must define keys for each of the custom `Connection` or `Statement` properties or attributes for which you would like to add support. For each custom key, create a `Simba::Support::AttributeData*` to store data for the property or attribute. Use a map to map keys to their corresponding `AttributeData*`. For more information on creating a custom key refer to the `DSIConnProperties.h` or `DSIStmtProperties.h` header files in the folder `[INSTALL_DIRECTORY]\DataAccessComponents\Include\DSI`.

To add custom connection and statement properties, implement the following methods in your `CustomerDSIIConnection` and/or `CustomerDSIIStatement` class:

- **IsCustomProperty()**

  In this function, check if the provided key corresponds with one of the standard ODBC properties. Return `false` if it does not; `true` otherwise.

  To see the list of keys for ODBC properties, see the `Simba::DSI::DSIStmtPropertyKey` enum or the `Simba::DSI::DSIConnPropertyKey DSIConnPropertyKey` enum in the C++ API reference. Go to Simba SDK C++ API Reference, select the **Namespaces** tab, select **Simba::DSI**, then search for **DSIConnPropertyKey**.

- **SetCustomProperty()**

  In this function, set an `AttributeData*` for the custom property key. In your implementation, check to ensure the provided key corresponds to a custom property or attribute. If it does not, an appropriate error or exception should be thrown and logged.

- **GetCustomProperty()**

  This function retrieves the `AttributeData*` associated with a custom key. In your implementation, check to ensure the provided key corresponds to a custom property or attribute.

- **GetCustomPropertyType()**

  This function retrieves the data type associated with the custom property or attribute. Data types are defined in the `Simba::Support::AttributeType` enum, located in the header folder `[INSTALL_DIRECTORY]\DataAccessComponents\Include\Support\AttributeData.h`.

Custom Properties in the Java SDK

Custom properties can be added to the connectors using the Java DSI with either the JNI DSI API, or the SimbaJDBC component. When using the JNI DSI API, custom properties are accessed in the same way that custom properties are accessed for ODBC connectors. When using the SimbaJDBC component, the custom properties are exposed through the following custom extensions to the `Connection` and `Statement` objects:

- **getAttribute(int)**

  Retrieve a custom property identified by the integer key.

- **setAttribute(int, Object)**

Set a custom property identified by the integer key.

Note:

Because these are custom extensions, applications will have to be coded to explicitly use these functions.

Custom Properties in the DotNet SDK

Custom properties can be added to connectors using the DotNet DSI, but can only be directly accessed when using the CLI DSI to build an ODBC connector.

Handling Connections

The ODBC application, the Simba ODBC layer, and your custom DSII layer interact to establish a connection to your data store. An important part of this process is obtaining all the required connection settings. The Simba SDK provides functions to help you manage the set of required and optional connection settings, and to repeat the request for settings until all required settings are obtained.

For a description of the connection process, see "Understanding the Connection Process" on the next page below.

Obtaining Settings and Connecting to the Data Store

In your `CustomerDSIIConnection::UpdateConnectionSetting` method, the `in_connectionSettings` parameter includes the connection settings that the user specified in the connection string, DSN, and/or prompt dialog. Your implementation of this method should return any modified or additional required (or optional) connection settings in the `out_connectionSettings` parameter.

You can use the utility functions `VerifyOptionalSetting` and `VerifyRequiredSetting` to help you check if a setting exists. If a setting does not exist, these functions put the appropriate value in the `out_connectionSettings` map.

To specify a list of acceptable values for one of your connection settings in the `out_connectionSettings` map, you must enter it yourself. For example:

```
DSIConnSettingRequestMap::const_iterator itr = in_connectionSettings.find(L"SomeSetting");

if (itr == in_connectionSettings.end())

{

    // Missing the required key, so add it to the requested settings.

    AutoPtr<ConnectionSetting> reqSetting(new

    ConnectionSetting(SETTING_REQUIRED));

    reqSetting->SetLabel(L"SomeSetting");

    reqSetting->RegisterWarningListener(GetWarningListener());

    std::vector<Simba::Support::Variant> values;

    values.push_back(Variant(L"YES"));

    values.push_back(Variant(L"NO"));

    values.push_back(Variant(L"UNKNOWN"));

    reqSetting->SetValues(values);

    out_connectionSettings[L"SomeSetting"] = reqSetting.Detach();

}
```

If `out_connectionSettings` contains additional required connection settings, then the Simba ODBC Layer will call `PromptDialog` to request these settings. Connection settings can be required or optional. This retrieve-request cycle repeats until all required connection settings have been provided. Once all required settings have been provided (even if some optional settings have not been provided), then the Simba ODBC Layer will call your `CustomerDSIIConnection::Connect` function.

Your implementation of the `CustomerDSIIConnection::Connect` function should establish a connection to your data store. You should inspect the `in_connectionSettings` parameter to retrieve any connection settings that are needed to establish and set up a connection to your data store. You can use the utility functions `GetOptionalSetting` and `GetRequiredSetting` to help you extract the settings from the `in_connectionSettings` parameter.

When your implementation of the `CustomerDSIIConnection::PromptDialog` function is called, you have the option of displaying a graphical dialog box to the user for requesting parameters or other connection settings. See "Creating and Using Dialogs" on the next page for more information about creating dialog boxes.

For example, if you require the user to enter a user id and a password, you can request those parameters from the user using this dialog box. If you do not wish to implement a dialog box, you can simply leave the `PromptDialog` function empty.

Example: Handling a Missing Password

Assume your DSII requires a user ID and password to establish a connection to your data store. Then, an application attempts a connection using `SQLDriverConnect` supplying the user ID setting but missing the password setting.

First, `UpdateConnectionSettings` is called so that all the settings that are needed for a connection can be verified. Your `UpdateConnectionSettings`f unction would use `VerifyRequiredSetting` for both the user ID and password keys to verify that they are present.

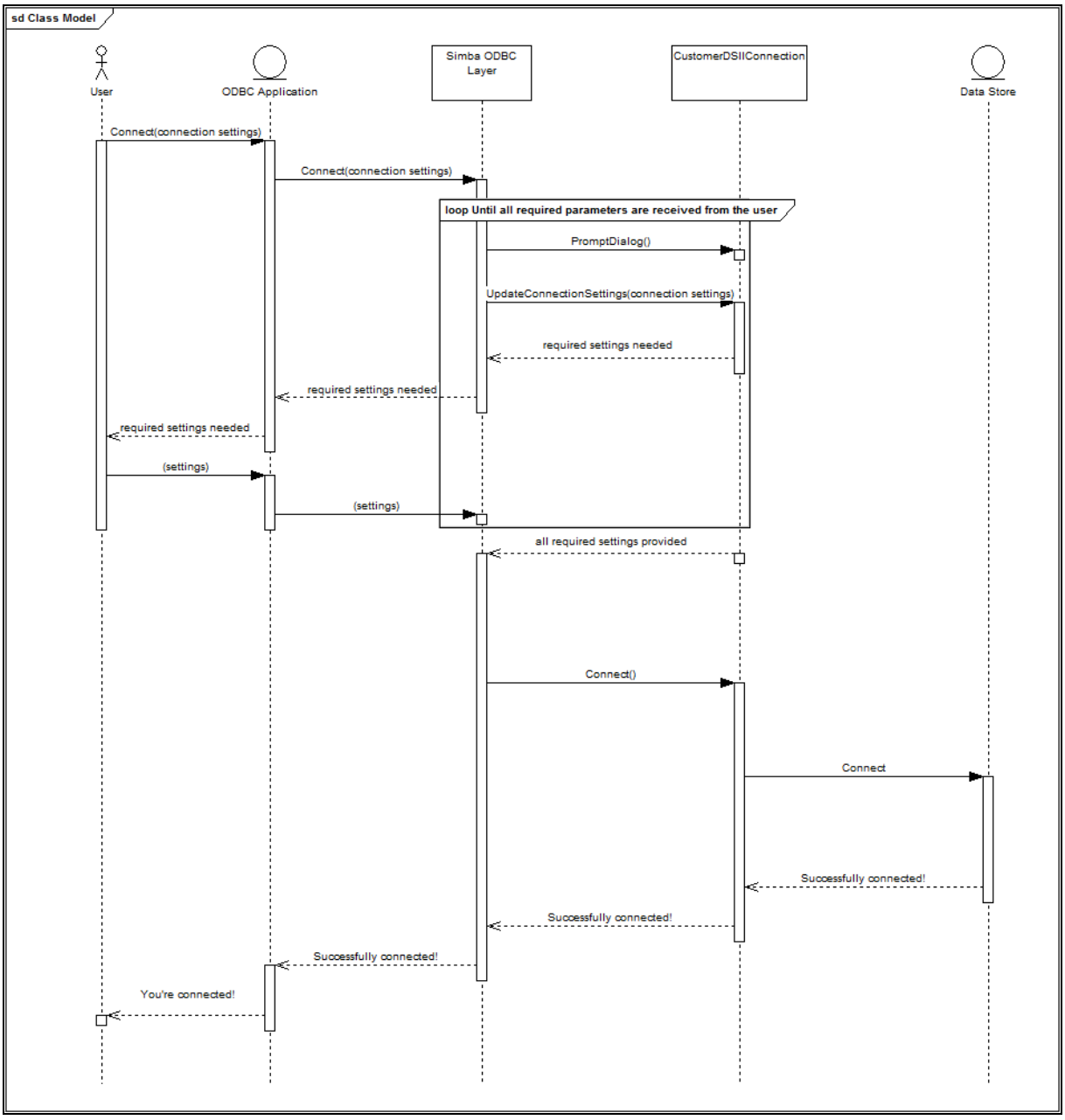
If any key is not present, it will be added to the `out_connectionSettings` parameter by `VerifyRequiredSetting`. Since the password key is missing, `out_connectionSettings` now contains that setting, and `UpdateConnectionSettings` will return.

When the Simba ODBC layer detects that a required setting is missing, it calls `PromptDialog`. This allows your DSII to prompt a dialog to the user to request any additional or missing information. Once the user has filled out the dialog and returned, the Simba ODBC layer will call `UpdateConnectionSettings` again to verify that all the required settings are now present. If all the required settings are present, it will then call `Connect` to proceed with the connection. If all the required settings are not present, it continue the `PromptDialog` and `UpdateConnectionSettings` cycle until the user cancels the dialog.

Understanding the Connection Process

This section provides a detailed explanation of how the end user, your ODBC-enabled application, the Simba ODBC Layer and your DSII layer interact to establish a connection to your data store. It then explains how you must handle the connection process in your own custom connector.

When the end-user initiates a connection to your data store, the Simba ODBC Layer will call your `CustomerDSIIConnection::UpdateConnectionSettings` function. Note that in some cases the Simba ODBC Layer may call your `CustomerDSIIConnection::PromptDialog` function, discussed later in this section, first if the connection parameters indicate it should do so. This process is shown in the diagram below:



Related Topics

"Creating and Using Dialogs" below

Creating and Using Dialogs

The Simba SDK includes functionality to help you implement dialogs. You can use these dialogs to retrieve user input such as connection settings or configuration information.

Dialogs in Windows

The Quickstart sample connector for Windows platforms includes a sample implementation of a user dialog. For information on the Quickstart sample connector, see *Build a C++ ODBC Connector in 5 Days* at http://www.simba.com/resources/sdk/documentation/.

This section shows you how to use the `PromptDialog` method of your `CustomerDSIIConnection` class to display a dialog box that prompts the user for settings for this connection. For more information on the connection process, see "Handling Connections" on page 41.

The `CustomerDSIIConnection::PromptDialog` method has the following signature:

```
virtual bool PromptDialog(
Simba::DSI::DSIConnSettingResponseMap& in_connResponseMap,
Simba::DSI::DSIConnSettingRequestMap& io_connectionSettings,
HWND in_parentWindow,
Simba::DSI::PromptType in_promptType
);
```

This method has the following parameters:

- **in_connResponseMap**

  The connection response map updated to reflect the user's input.

- **io_connectionSettings**

  The connection settings map updated with settings that are still needed and were not supplied. The connection settings from io_connectionSettings are presented as key-value string pairs. The input connection settings map is the initial state of the dialog box. The input connection settings map will be modified to reflect the user's input to the dialog box.

- **in_parentWindow**

  Handle to the parent window to which this dialog belongs.

- **in_promptType**

  Indicates what type of connection settings to request either both required and optional settings or just required settings.The return value for this method indicates if the user completed the process by clicking OK on the dialog box (return true), or if the user aborts the process by clicking CANCEL on the dialog box (return false).

Linux/Unix/macOS

Dialogs are also possible on Linux/Unix/macOS platforms, although the Quickstart sample connector for those platforms does not include a sample implementation.

The `PromptDialog` function is the same as for Windows. However, the meaning of the `in_parentWindow` argument is undefined. Different applications may potentially pass in different types of window handles. Therefore, `in_parentWindow` can only be used if your connector can make assumptions about running within a specific window system or API toolkit. Otherwise, the window you create will need to be parentless.

Related Topics

"Handling Connections" on page 41

Canceling Operations

Prior to ODBC 3.8, only statement operations could be cancel using `SQLCancel`. In ODBC 3.8 a new function called `SQLCancelHandle` was added that can cancel both statement and connection operations. Note that canceling a statement in 3.8 using `SQLCancelHandle` is identical to canceling it using `SQLCancel`.

Simba SDK supports both `SQLCancelHandle` and `SQLCancel`. The implementations of `DSIConnection` and `DSIStatement` can handle and clear the cancel requests through the `OnCancel` and `ClearCancel` callbacks. The following table summarizes this functionality:

Class	OnCancel	ClearCancel
DSIConnection	Invoked when SQLCancelHandle is called on the `DSIConnection`'s handle.	Invoked at the beginning of a connection related function that has the ability to be canceled.
DSIStatement	Invoked when `SQLCancelHandle` or `SQLCancel` is called on the `DSIStatement`'s handle.	Invoked at the beginning of a statement function that has the ability to be canceled.

In `OnCancel`, the object can perform any cancellation logic such as setting flags to indicate that an operation should be canceled.

In `ClearCancel`, the object can clear any pending cancel notification (e.g. clear flags).

Handling Transactions

A transaction is a set of operations that are executed on a data store. If a transaction is successful, all of the data modifications made during the transaction are committed. If a transaction encounters errors and must be canceled or rolled back, then all of the data modifications are erased.

If your data store supports transactions, you can enable them in your custom ODBC or JDBC connector. To enable transactions in your connector, your DSII must enable both read and write functionality.

To enable read/write capability on your connector:

- For the C++ SDK, call `DSIPropertyUtilities::SetReadOnly`, passing in `false` for the second parameter.
- For the Java SDK, call `PropertyUtilities::SetReadOnly`, passing in `false` for the second parameter.

Enabling Transaction Support

After adding read/write capability to your custom connector, you can enable transaction support by creating your own implementation of the `DSIIConection` class and implementing the `BeginTransaction()`, `Commit()`, and `Rollback()` methods.

Implement the DSIConnection Class

Support for transactions is implemented in the `DSIConnection` class. Override this class so you can provide your own implementation.

Specify that Transactions are Supported

The Simba SDK uses a property to specify the level of transaction support for a custom connector. This is done differently in C++ and in Java.

To set the DSI_CONN_TXN_CAPABLE Property in C++:

Set the DSI_CONN_TXN_CAPABLE property in your `DSIConnection` object to specify the level of transaction support that your connector can handle. You can use the `DSIPropertyUtilities::SetTransactionSupport()` helper method.

Example: Setting the DSI_CONN_TXN_CAPABLE property in C++

In this example, a helper method called `SetConnectionPropertyValues()` is used to set the DSI_CONN_TXN_CAPABLE property. This method is invoked from the `MyConnection` class's constructor. It calls the `SetReadOnly` method, passing in `false`:

```
void MyConnection::SetConnectionPropertyValues(){
 DSIPropertyUtilities::SetReadOnly(this, false);
 DSIPropertyUtilities::SetTransactionSupport(this, DSI_TC_DML);
        ...
```

In the above example, transaction support is set to DSI_TC_DML, which only supports DML statements within a transaction.

To set the DSI_TXN_CAPABLE Property in Java:

Set the ConnPropertyKey.DSI_TXN_CAPABLEin your `DSIConnection` object to specify the level of transaction support that your connector can handle. You can do this using the `DSIConnection::SetProperty()` method, passing DSI_TXN_CAPABLE as the attribute data.

Implement the Required Methods in the DSIConnection Class

To support transactions, your connector must implement the methods as described in this section.

1. **BeginTransaction()**

   This method is invoked by the Simba SDK at the start of a new transaction on the connection. This method is responsible for performing any logic that is required before the transaction starts, such as ensuring that transactions are supported, or checking that a transaction is not already in progress.

   Example: Check whether a transaction is already in progress

   In this example, the custom connector checks a member variable that tracks whether a transaction is already in progress. If so, an exception is thrown. Otherwise, the member is set to `true` at this

point. Subsequent transaction methods will check this variable to coordinate their workflows with the current transaction.

```
void MyConnection::BeginTransaction() {
  if (isInTransaction) {
    XMTHROWGEN(" Illegal transaction state change(BeginTransaction). ");
  }
  else {
    isInTransaction = true;
  }
}
```

2. **Commit()**

This method is invoked by the Simba SDK to commit the statements of a transaction. This method is responsible for performing commit-related logic for the outstanding transaction on the connection, such as storing any inserted or updated data.

Example: Sample Commit() Implementation

This example shows one way that you could implement your `Commit()` method. It also shows a helper method, `MyConnection::CommitImp()`.

```
void MyConnection::Commit(){
    if (!isInTransaction) {
        XMTHROWGEN(" Illegal transaction state change(Commit). ");
    }
    else {
        isInTransaction = false;
        CommitImpl();
    }
}
////////////////////////////////////////////////////////////////////////////////////////////////////
void MyConnection::CommitImpl() {
    // Get the Connector's Tables
    AutoValueMap<XMTableIdentifier, const XMTableData>& driverTableDataMap =
    GetTableDataMap();
    CriticalSectionLock lock(m_criticalSection);
```

```
        for (AutoValueMap<XMTableIdentifier, const XMTableData>::iterator it = m_
        changedTables.begin(); it != m_changedTables.end(); it++){
                MapUtilities::InsertOrUpdate(driverTableDataMap, it->first, it->second);
                it->second = NULL;
        }
}
```

`MyConnection::Commit()` first ensures that a transaction is in progress, and then delegates the commit logic to `MyConnection::CommitImpl`. `MyConnection::CommitImpl` first takes a lock. Then, it iterates through its `m_changedTables` member, which is used to track tables that have had inserts or updates to made to their values. Finally, it uses the `MapUtilities::InsertOrUpdate` helper method to perform the actual data value insertion/updates on these tables.

3. **Rollback()**

This method is invoked by the Simba SDK when a ROLLBACK statement is encountered in a transaction query. This method is responsible for rolling back data for an outstanding transaction on the connection, to the state it was in before the start of the transaction.

Example: Sample Rollback() Implementation

This example shows one way that you could implement your `Rollback()` method.

```
void XMConnection::Rollback(){
        if (!isInTransaction){
                XMTHROWGEN(" Illegal transaction state change(Rollback). ");
        }
        else {
                isInTransaction = false;
                m_changedTables.DeleteClear();
        }
}
```

This method first ensures that a transaction is in progress. If the transaction is in progress, it resets `isInTransaction` to indicate that a transaction is no longer in progress, followed by a call to `m_changedTables.DeleteClear()`, which clears the listing of tables that have been modified.

Adding Support for Savepoints (SimbaJDBC only)

A *savepoint* is a way of implementing subtransactions, which are also called *nested transactions*. A savepoint is used to mark a point in a transaction that you can roll back to without affecting any work done in the transaction before the savepoint was created. Savepoints are supported for DSIIs that are written in Java and use the SimbaJDBC component.

Set the DSI_SUPPORTS_SAVEPOINTS property

Set the DSI_SUPPORTS_SAVEPOINTS property in your custom `DSIConnection` object to DSI_SUPPORTS_SAVEPOINTS_TRUE.

Implement the Required Methods in the DSIConnection Class

Modify your custom `DSIConnection` object to override and implement the following virtual methods:

1. **createSavepoint(String)**

This method is invoked by the Simba SDK when a SAVEPOINT statement is encountered in a query. This method is responsible for creating a new Savepoint with the specified name in the current transaction, and performing any save point logic such as caching information about the current state of data, that could be used to restore the state if a subsequent rollback operation occurs.

2. **releaseSavepoint(String)**

This method is invoked by the Simba SDK when a RELEASE statement is countered in a query. This method is responsible for releasing the Savepoint with the specified name so that the Savepoint is no longer available to rollback to. This could also include performing any related logic such as freeing up resources and clearing any state information that was related to the specified save point.

3. **rollback(String)**

This method is invoked by the Simba SDK when a ROLLBACK statement is encountered in a transaction query. This method is responsible for rolling back data for an outstanding transaction on the connection, to the state it was in before the start of the transaction.

Supporting Transactions through SQL

In some data sources, transactions can also be triggered by executing certain SQL queries (e.g. BEGIN, COMMIT, AND ROLLBACK statements). Support for this is provided through the `ITransactionStateListener` interface, which allows your DSII to inform the Simba components of any changes in transaction state.

In your `CustomerDSIConnection` object, invoke the following methods on the `m_transactionStateListener` member exposed by the `DSIConnection` class. This informs the Simba components of any changes to transaction state.

In the C++ SDK:

1. When a transaction has started, call `ITransactionStateListener::NotifyBegin`.
2. When a transaction is committed, call `ITransactionStateListener::NotifyCommit`.
3. When a transaction is rolled back, call `ITransactionStateListener::NotifyRollback`.

In the Java SDK:

1. When a transaction has started, call `ITransactionStateListener.NotifyBeginTransaction`.
2. When a transaction is committed, call `ITransactionStateListener::NotifyCommit`.
3. When a transaction is rolled back, call `ITransactionStateListener::NotifyRollback`.

If your DSII is written in Java and is using the SimbaJDBC component, then you may need to notify the `ITransactionStateListener` about Savepoint operations as well:

1. When a Savepoint is created, call `ITransactionStateListener::notifyCreateSavepoint`.
2. When a Savepoint is released, call `ITransactionStateListener::notifyReleaseSavepoint`.
3. When a transaction is rolled back to a Savepoint, call `ITransactionStateListener::notifyRollbackSavepoint`.

> **Important:**
> **Important:**
>
> ODBC does not support Savepoints, and attempting to use the `notify*Savepoint`functions on the `ITransactionStateListener` while using the JNI DSI API will cause an exception.

Bulk Fetch in the C++ SDK

Prior to Simba SDK 10.0, data had to be retrieved from an `IResult` row by row and column by column using the `Move` method to position the cursor, and `RetrieveData` to return a cell of data. Retrieving data in this manner is acceptable for small-to-medium sized datasets, or those with results spanning non-contiguous rows, but has the following drawbacks:

- Data is accessed per cell, which means the ODBC layer of the SDK needs to loop through each row and each column, invoking methods to retrieve and convert each individual cell of data. This results in a large number of small data transfers, with each of them requiring a small amount of overhead, but collectively resulting in a noticeable impact on performance.
- The retrieval of each data cell involves invoking multiple virtual methods, which can stall a CPU's instruction pipeline and decrease a connector's execution performance.

As of 10.0, `IResult` now exposes the Bulk Fetch API which provides a more optimized data retrieval mechanism allowing a connector to fetch contiguous rows of data via a single method call and store the data directly into the buffer allocated by the calling application. This "bulk fetch" mechanism eliminates the need to iterate over rows and columns to return data and allows all the data for many rows to be returned in one pass.

Note:

- Bulk fetch is currently supported for ODBC connectors that do not use the SQL Engine and are not implemented by the Simba SDK's ODBC Client. Bulk Fetch can be implemented in SimbaServer, as described below.
- Bulk fetch is supported in the C++ SDK only.

Overview

This section describes the high-level overview for Bulk Fetch.

1. The ODBC layer of the SDK instantiates one Bulk Processor for each column bound by the application. A Bulk Processor is an object that oversees the process of copying and converting

contiguous rows of data for a bound column. Each Bulk Processor contains all the necessary information about the data buffer and length/indicator field that the application has bound to the column as well as a Bulk Converter which is an object that is able to convert values from the SQL data type (the data type that the DSII uses to talk to the SDK) to a C data type (the data type returned to the application).

2. The ODBC layer calls the BulkFetch method implementation of the DSII.
3. The DSII retrieves multiple rows for all the bound columns. Note that it can also retrieve the data of other columns, but they won't be used. The DSII can organize the data as it wants in memory, but needs to leave it unchanged until the bulk conversion is complete.
4. The DSII needs to instantiate one Column Segment per bound column. A Column Segment describes where in memory the contiguous set of data rows for a column can be found as well as the offsets required to find the next row. The Bulk Processor uses a Column Segment so it knows where the values of the retrieved rows have been stored in memory.
5. The DSII instructs the Bulk Processors to convert the columns (i.e. perform the bulk fetch), providing them with the Column Segments it has just instantiated for each bound column. The Bulk Processors convert all the SQL values retrieved by the DSII to the C values directly into the buffer bound by the application.
6. Once the bulk fetch completes, the ODBC layer of the SDK does not need to perform any further data processing because the Bulk Fetch has both converted and copied the data for all rows to be returned directly to the application's buffer.

Since the Bulk Fetch API writes directly to the buffer provided by the application, bulk fetch functionality can only be used for columns which are bound by the calling application (i.e. columns for which memory has been allocated and associated with each column to store the data returned by the connector) and of those, only for columns which have been selected for retrieval (i.e. those columns specified in a SELECT query). For efficiency reasons, the SDK also uses the Bulk Fetch API only when the application requests multiple rows during each fetch (when the size of the row set being requested contains at least two rows).

In its first release, the Bulk Fetch API is only supported for DSII's that do not rely on the SEN SDK SQLEngine for query execution. Performance is gained with "bulk fetch" when multiple rows of bound columns are accessed and converted sequentially. The SQLEngine on the other hand, gets the value of cells (the value of a specific column of a specific row) on demand and might apply a different data conversion for each cell value. This does not fit the Bulk Fetch API design.

In addition to the performance benefits listed above, bulk fetch also provides the following advantages:

- Bulk Processors implemented in the SDK are independent of each other. A DSII can therefore create one thread per bound column and do the bulk conversion of all the columns in parallel. This could also include the retrieval of data if the rows of the various columns are independent of each other (e.g. if the data store has a columnar organization).

- Bulk Converter factories are created by the connection. The factory objects which create the Bulk Converters are instantiated by the DSII's connection object (via `IConnection::GetSqlToCBulkConverterFactory()`) which means the connection can determine the type of factories to create (for example, a connection could return a factory type based on the type of server it is connecting to). This provides more flexibility than the singleton converter factory used under normal (non bulk fetch) data retrieval which forces all connections to use the same type of converter.

Bulk Fetch API

This section describes the objects and methods that are related to the bulk fetch API.

Methods in IResult

These methods, defined in IResult, make up the bulk fetch API. These methods must be implemented in your connector's `IResult` class.

IsBulkFetchSupported()

This method is invoked by the Simba ODBC layer before attempting a bulk fetch, in order to determine if bulk fetch is supported. This method takes in the indices of the bound columns for which data is being selected, and allows the connector to tell the Simba ODBC layer whether or not it can support bulk fetching for those columns.

If bulk fetch is not supported by your connector, then return false for `IsBulkFetchSupported`. In the `DSISimpleResultSet` class provided by the SDK, this method returns false and must be overridden to return true if your connector supports bulk fetch. This class also defines a simple implementation for the `BulkFetch` method which positions the cursor and then invokes a `DoBulkFetch()` method which must be overridden to perform the bulk fetch.

BulkFetch()

This method is invoked by the Simba ODBC layer to perform a bulk fetch, when an application requests rows for bound columns. This method takes in the number of rows to return and a collection of `IBulkProcessors`. An `IBulkProcessor` converts the data for multiple rows of a bound column and stores it directly into the buffers bound to those columns by the application. Additional detail is provided below.

Additional Classes and Methods

This section summarizes the additional interfaces and classes that your connector will use to support bulk fetch functionality. Note that default implementations are provided for each. Additional detail for each of these components is provided later in this section.

IBulkProcessor

This interface defines the interface for a Bulk Processor. The ODBC layer of the SDK provides an implementation called `SqlToCBulkConverterWrapper`. `SqlToCBulkConverterWrapper` delegates the copy-and-conversion process to an `ISqlToCBulkConverter` (described below).

AbstractColumnSegment

A concrete implementation of this class is constructed by the `IResult` object when the `BulkFetch` method is invoked by the Simba ODBC layer. The `IResult` object will then pass this object to the `IBulkProcessors` for use in converting column data from the data source. The SDK provides the following two default concrete implementations, although applications can also implement their own:

- `FixedRowSizeColumnSegment`: suitable for use when the underlying data to be retrieved is stored in a buffer in which a fixed number of bytes are allocated per cell and the address of the cell for each successive row can be computed by adding a constant offset to the memory pointer.

- `DataLengthColumnSegment`: suitable for use when the data to be retrieved is stored in a buffer in which a variable number of bytes are allocated per cell, and therefore the address of the next cell cannot be computed using a constant offset. This class stores a collection of DataLengthColumn objects each of which describes the memory location and length of data for a particular cell.

- `ServerColumnSegment`: must be used when implementing bulk fetch on the server. It is designed to optimize performance over the SimbaClient/Server wire protocol. This class takes the following arguments:
  - `in_data` - An array of pointers to the data
  - `in_lengths` - an array specifying the length of each item of data in in_data. The array value must be -1 if the corresponding data has no length (is NULL). If the corresponding data is fixed-length type, the array value is not used (but must be a non-negative integer). If the corresponding data is a variable-length type, the array value must specify the length of the data, in bytes.
  - `in_count` - a counter specifying the length of in_data and in_lengths.

Note:

Using Bulk Fetch in SimbaClient/Server may cause issues in exposing warnings and errors associated with fetching ResultSet data. When data is fetched row by row, the error can be associated with a specific cell in the data, and is returned to the client when the cursor moves to the next row. With Bulk Fetch however, the error is returned to the client along with the bulk chunk of data, and the error is not associated with a specific cell.

ISqlToCBulkConverter

Defines an interface for a Bulk Converter. The SDK provides a default implementation called `SqlToCBulkConverter` which is (derives from) a templated functor and performs a conversion from a SQL data type to a C data type. The SDK also #defines hundreds of templated functor operator() methods for conversions of specific data types. Although the use of templates per converter increases the size of the compiled binary connector (i.e. a `SqlToCBulkConverter` class will be defined by the compiler for each #defined template), it eliminates the need to subclass a converter for each possible data type conversion and therefore eliminates virtual calls.

ISqlToCBulkConverterFactory:

Creates the `ISqlToCBulkConverter` object that will be used by the `IBulkProcesser` object to copy and convert data for a specific column. The SDK provides a default implementation called `DSIBulkConverterFactory` which, through templates, determines the correct `ISqlToCBulkConverter` to return to handle the data type of the column.

IConnection::GetSqlToCBulkConverterFactory

Invoked by the Simba ODBC layer on a connection to obtain the `ISqlToCBulkConverterFactory` object that will be used to construct `theISqlToCBulkConverter` objects for each `SqlToCBulkConverterWrapper`. The Simba ODBC layer will create a `SqlToCBulkConverterWrapper` for each column, passing the `ISqlToCBulkConverterFactory` to the constructor.

Settings

When supporting bulk fetch, you can allow your customers to configure additional settings at runtime. This step is optional. For example, the following settings can be configured in the registry or through a connection string:

- `UseBulkFetch`: set to 1 to enable bulk fetches, or 0 to disable.
- `UseSqlEngine`: must be set to 0 when enabling bulk fetches. Currently the SQL Engine cannot be used with bulk fetches.
- `ColumnSegmentId`: specifies the Column Segment class that should be used to provide information about the buffer bound by the application. Set to 1 to use `FixedRowSizeColumnSegment`, 2 for the `DataLengthColumnSegment`, or the ID of your custom Column Segment class (see "Creating a Custom Column Segment and Converter" on page 61 for more information).

One way to implement these runtime setting is to pass them in to the connector's `Connection::Connect()` method. The method then passes the `UseBulkFetch` and `ColumnSegmentId` values to the table during construction. The table then uses these values to determine if bulk fetch is supported, and which Column Segment type to use.

Adding Bulk Fetch to a Connector

To add bulk fetch to your connector, we recommend subclassing `DSISimpleResultSet`. This is the quickest way to implement your IResult, and provides easy access to the default implementations provided by the Simba SDK.

This section provides code samples that you can use when subclassing `DSISimpleResultSet` and adding bulk fetch to your connector.

When subclassing `DSISimpleResultSet`, the first method to override is `IsBulkFetchSupported()`.This method is invoked by the Simba ODBC Layer for each bound column in the bulk fetch, and provided with the column index. Your connector will use this method to signal to the Simba ODBC layer whether bulk fetch is supported for each column in the table from which data is being requested by a bulk fetch. If your connector does not support bulk fetch or if bulk fetch is disabled (e.g. through a setting), then this method should always return false. The following code snippet shows the implementation from an example `DSISimpleResultSet`-derived class, `XMTableLight`.

```
bool XMTableLight::IsBulkFetchSupported(std::set<simba_uint32>& in_boundColumnIndex)
{
  UNUSED(in_boundColumnIndex);
  return m_useBulkFetch;
}
```

This example shows the latter case where bulk fetch can be enabled or disabled. The class stores a flag (`m_useBulkFetch`) which is set in the class's constructor based on the settings provided to it from the Simba ODBC layer (i.e. a flag indicating whether or not the user enabled bulk fetch). `IsBulkFetchSupported` then returns this flag to the ODBC layer regardless of the column index. In your connector, you will most likely want to examine the column metadata corresponding to the column indexes provided in `in_boundColumnIndex` and then decide whether or not bulk fetch is supported for all corresponding columns for the query currently under execution.

Note:

When deriving a result set from `DSISimpleResultSet`, the default implementation returns `false`, so `DSISimpleResultSet`-derived classes don't have to do anything if a connector doesn't support bulk fetch. However, a connector which directly implements IResult must implement this method and return false if it doesn't support bulk fetch.

The next method to implement is `BulkFetch`. `DSISimpleResultSet` provides a `BulkFetch` implementation which keeps track of the current row, and delegates the bulk fetch logic to a protected method called `DoBulkFetch` that your connector must implement. The following snippet shows the `BulkFetch` implementation provided by `DSISimpleResultSet`:

```
simba_unsigned_native DSISimpleResultSet::BulkFetch(
  simba_unsigned_native in_rowsetSize,
  const std::vector<Simba::DSI::IBulkProcessor*>& in_bulkProcessors)
{
  if (!m_hasStartedFetch)
  {
    // Go to the first row.
    m_hasStartedFetch = true;
    m_currentRow = 0;
  }
  else
  {
    // Move on to the next row.
    m_currentRow++;
  }
  const simba_unsigned_native rowsFetched(DoBulkFetch(in_rowsetSize, in_bulkProcessors));
  if (rowsFetched > 0)
  {
    m_currentRow += (rowsFetched - 1);
  }
  return rowsFetched;
}
```

`BulkFetch` takes in the number of rows to obtain along with the collection of bulk processors to use for each column. Note that in default implementation, the Simba ODBC layer will pass a collection of `SqlToCBulkConveterWrapper` objects.

In `DSISimpleResultSet`'s implementation, the class manages an `m_currentRow` member, which is the index of the current row to obtain data from. Since multiple bulk fetches can be invoked where each obtains a limited number of rows, this method begins by checking `m_hasStartedFetch` to determine if a previous bulk fetch has been made. If not (i.e. this is the first bulk fetch or the cursor has been

closed), `m_currentRow` is set to the first row, otherwise, it is advanced to the next row in preparation of the bulk fetch. The method then delegates the bulk fetch logic to the `DoBulkFetch` method, and forwards the parameters to that method. `DoBulkFetch` performs the bulk fetch returning the number of rows fetched. `BulkFetch` then advances `m_currentRow` by the number of rows fetched and adjusts it (subtracts 1) since it is zero based.

For example, the `XMTableLight` class derives from `DSISimpleResultSet` and provides the implementation of `DoBulkFetch`. The following code snippets break down the main parts of this method:

```
simba_unsigned_native XMTableLight::DoBulkFetch(
   simba_unsigned_native in_maxRows,
   const std::vector<IBulkProcessor*>& in_bulkProcessors)
{
   const simba_unsigned_native firstRow(GetCurrentRow());
   if (firstRow >= m_totalRows)
   {
      return 0;
   }
   const simba_unsigned_native rowsToReturn(simba_min(in_maxRows, m_totalRows - firstRow));
   const AutoVector<XMTableColumnDataBase>& columns(m_tableData->GetDataCol());
   AutoVector<IBulkProcessor>::const_iterator it(in_bulkProcessors.begin());
   const AutoVector<IBulkProcessor>::const_iterator end(in_bulkProcessors.end());
```

This method begins by ensuring that `m_currentRow` has not been advanced passed the end of the rows in the table. If it has (i.e. all rows have been fetched), then the method returns 0 to indicate that no more rows are available to be fetched. The method then determines the number of rows that will be returned by determining the lower value of the number of rows remaining or the number of rows requested for bulk fetch.

After this, the method prepares a collection of `XMTableColumnDataBase` objects which provide access to the underlying data for each column. It then constructs an iterator that will be used to iterate over each of the bulk processors passed to the method by the Simba ODBC SDK, and perform the bulk fetches.

The next snippet shows the core loop where the method performs this iteration. The purpose of this loop is to invoke the bulk fetch process on each column bound by the application. This is accomplished by iterating through each Bulk Processor passed in from the Simba ODBC layer, constructing the appropriate Column Segment object based on the configuration settings, and invoking the Process method on the current Bulk Processor, passing in the newly-constructed Column Segment describing where the table data can be found.

```
for (; it != end; ++it)
{
        IBulkProcessor& processor(**it);
```

```
        const SelectListItem& item(GetSelectListItem(processor.GetColumnIndex()));
        const XMTableColumnDataBase& column(*columns[item.first]);
        RightTrimmer* const rightTrimmer(item.second ? m_rightTrimmers[item.first] : NULL);
        switch (m_columnSegmentId)
        {
                case AbstractColumnSegment::FIXEDROWSIZE_ID:
                ... do fixed row size processing (see below)
                break;
                case AbstractColumnSegment::DATALENGTH_ID:
                ... do data length row processing (see below)
                break;
                case XMStringColumnSegment::XM_COLUMNSEGMENT_ID:
                ... do row processing using a custom Column Segment (see below)
                break;
        }
}
```

The loop starts by using the column index reported by the current Bulk Processor to determine if the column data should be right trimmed (a typedef called `SelectListItem` stores the index and a flag) and then obtains a reference to the `XMTableColumnDataBase` object corresponding to the column index of the current Bulk Processor, which contains the underlying data for the column.

A switch/case statement is then used to determine which type of Column Segment to construct, based on the application settings:

```
switch (m_columnSegmentId)
{
        case AbstractColumnSegment::FIXEDROWSIZE_ID:
        {
                std::vector<std::pair<const void*, simba_uint32> > sourceBuffers
                (rowsToReturn);
                simba_uint32 maximumDataSize = 0;
                for (simba_signed_native index = 0; index < rowsToReturn; index++)
                {
                        const RowIdentifier& rowId = m_rows[firstRow + index];
                        sourceBuffers[index] = column.GetBuffer(rowId);
```

```
        if (maximumDataSize < sourceBuffers[index].second)
        {
                maximumDataSize = sourceBuffers[index].second;
        }
}
std::vector<simba_byte> cachedDataBuffer(maximumDataSize *
rowsToReturn);
std::vector<simba_signed_native> cachedLengthBuffer(rowsToReturn);
simba_byte* cellPtr = &cachedDataBuffer[0];
for (simba_signed_native index = 0; index < rowsToReturn; index++, cellPtr
+= maximumDataSize)
{
        const std::pair<const void*, simba_uint32>& sourceBuffer =
        sourceBuffers[index];
        if (NULL != sourceBuffer.first)
        {
                memcpy(cellPtr, sourceBuffer.first,
                sourceBuffer.second);
                cachedLengthBuffer[index] =
                sourceBuffer.second;
        }
        else
        {
                cachedLengthBuffer[index] =
                CvtLength::MakeNull();
        }
}
FixedRowSizeColumnSegment columnSegment(
&cachedDataBuffer[0],
maximumDataSize,
&cachedLengthBuffer[0],
sizeof(simba_signed_native),
```

```
            rowsToReturn);
            processor.Process(columnSegment);
            break;
        }
        case AbstractColumnSegment::DATALENGTH_ID:
        {
            std::vector<DataLengthColumn> dataLengthColumns(rowsToReturn);
            for (simba_signed_native index = 0; index < rowsToReturn; index++)
            {
                const RowIdentifier& rowId = m_rows[firstRow + index];
                const std::pair<const void*, simba_uint32> columnData
                (column.GetBuffer(rowId));
                if (NULL == columnData.first)
                {
                    dataLengthColumns[index].SetAttributes(NULL,
                    CvtLength::MakeNull());
                }
                else
                {
                    dataLengthColumns[index].SetAttributes
                    (columnData.first, columnData.second);
                }
            }
            DataLengthColumnSegment columnSegment(&dataLengthColumns[0],
            rowsToReturn);
            processor.Process(columnSegment);
            break;
        }
        case XMStringColumnSegment::XM_COLUMNSEGMENT_ID:
        {
            // For the SQLite custom column segment, the conversion is specialized
            depending on the
```

```
            // column type. So delegate the conversion to the colum object.
            column.Process(processor, m_rows, firstRow, rowsToReturn,
            rightTrimmer);
            break;
        }
        ...
}
```

Fixed Row Size Processing

In the case of a `FixedRowSizeColumnSegment`, the Column Segment constructor takes in pointers to two buffers: one containing the underlying table data stored in a contiguous array of values, and the other containing the data length for each cell stored for the column.

Since this sample connector does not store its underlying table data contiguously, the code first iterates through each row starting at the first row identified above, fetching the cell value for the column along with the size of the data, and storing it in a temporary collection of data/length pairs. During this process it also identifies the maximum data size, and stores it in `maximumDataSize`. This is used further down to specify the offset for finding the cell for the next row, within the buffer.

The code then separates and caches these data/length pairs into the two separate buffers required by `FixedRowSizeColumnSegment`. When `FixedRowSizeColumnSegment` is being constructed, the buffers are passed into the constructor along with `maximumDataSize`, the size of the length values (which specifies the offset to find the next cell size value), and the number of rows to fetch. The segment uses `maximumDataSize` and the size of the length values so that it knows where to find the next element in to two buffers respectively, as it iterates through each row.

Finally, the Column Segment is passed to the Process method of the current Bulk Processor to bulk fetch the data for the column. Process will then use its internal converter to convert and copy each cell to the application buffer bound to the column. Since the Simba ODBC layer has already configured the Bulk Processor with the location of the application buffer, the Bulk Processor already knows where the data is to be copied to.

Note that this example is for demonstration purposes only. Since the sample connector doesn't store its data in the format required by the `FixedRowSizeColumnSegment` class, additional overhead in terms of processing and memory was necessary to cache the variable length data and sizes into the buffers expected by `FixedRowSizeColumnSegment`. Therefore, a better solution for this type of table data would be to use the `DataLengthColumnSegment` as described next.

Data Length Row Processing

In the case of `DataLengthColumnSegment`, a collection of `DataLengthColumn` objects are created for each cell from each row to return. `DataLengthColumn` is a helper class which describes the location and length for a single cell of data, and the `DataLengthColumnSegment` class requires a collection of these objects for each row to return, along with the total number of rows to return.

In the example snippet, the code iterates through each row, obtaining the row's ID and invoking `column.GetBuffer` to obtain the address where the cell's data is stored for that row. This address is stored a temporary pair object which takes in and stores the location of the cell data, and automatically computes the sizeof the data based on the second template parameter. Note that for simplicity, this example assumes that the column contains integer data. In your connector, it may be necessary to

compute or obtain the length of cell data based on the type of data stored in the column (e.g. variable length character data), rather relying on sizeof.

This information is then passed to the `DataLengthColumn` object via the `SetAttributes` method. After the collection of `DataLengthColumn` objects has been created, it's passed along with the row count to return to a new `DataLengthColumnSegment` which in turn, is passed to the Bulk Processor to perform the bulk fetch.

As this example shows, the use of `DataLengthColumnSegment` is a better solution for the sample `XMTableLight` class than `FixedRowSizeColumnSegment`, because `XMTableLight` stores its data in non-contiguous arrays and can easily and more quickly describe the location and size of each cell by simply populating a collection of `DataLengthColumn` objects.

Also note the use of the `CvtLength` class. This utility class provides methods which allow the DSII to encode and decode the length of the data before and after data conversion. This is necessary because the encoded length must be used when creating Column Segments. Bulk Converters use this length to detect cases when data is null or was not successfully retrieved from the data source. Custom Bulk Converters also need to use this class when setting the target length resulting from the conversion. The following list outlines the various cases that the `CvtLength` class handles:

- Normal length: the length of data that is not null and was successfully converted (no truncation required).
- Truncated length: the length of data was either truncated during data retrieval or data conversion.
- Null value: a null value was retrieved from the data source. Note that null values are not passed to the conversion functors in the `SqlToCBulkConverter` template that handles the SDK Column Segment implementations (see `SqlToCBulkConverter.h`). For optimization reasons, the Conversion Functors do not handle null values and most of them will assert in debug mode or generate an invalid value in release mode. Therefore the same must be done in the implementation of a custom Column Segment.
- Retrieval error: used if the value of a cell cannot be retrieved successfully from the data source. The default behaviour of the Column Segment implementations provided by the SDK is to generate a retrieval error diagnostic. A DSII could however decide to discard the row (not referencing it in the column segment) or terminate the Bulk Fetch operation. The recommended approach is to handle it in the same way as when encountering a retrieval error during a single cell fetch.

For more information about the various methods available, see `CvtLength.h`.

Row Processing using Custom Column Segment

The final case statement in the example, checks for a custom Column Segment ID and the delegates the Bulk Fetch to the `XMTableColumnDataBase` object's Process method which has been set up to use a custom Column Segment. Information on creating and using a custom Column Segment is provided next.

Creating a Custom Column Segment and Converter

The `FixedRowSizeColumnSegment` and `DataLengthColumnSegment` classes provided by the SDK can be used by most connectors for bulk fetch because they describe data in both fixed-length,

and variable-length storage respectively. However, developers are free to implement their custom Column Segment classes to improve efficiency or provide additional convenience in specifying where data is located.

For example you could implement your column segment and StringColumnSegment to provide direct access to your data address/length mappings, rather than requiring the DSII to copy pointers/lengths into an intermediate buffer.

The following steps describe how to create and use a custom Column Segment:

1. Derive a new class from `AbstractColumnSegment` for your custom Column Segment ensuring that at a minimum, the constructor takes in a number of type `simba_unsigned_native`, which will be used to specify the number of rows that are to be retrieved for a given bulk fetch. Additional parameters can also be added as required by your connector. For example, `XMColumnSegment` also takes in a reference to the underlying column data, a reference to the row ID's from which to obtain data, and the row number of the starting row:

   ```
   XMColumnSegment(
   const std::map<RowIdentifier, T>& in_columnData,
   const std::vector<RowIdentifier>& in_rows,
   simba_unsigned_native in_startRow,
   simba_unsigned_native in_numRows) :
   AbstractColumnSegment(XM_COLUMNSEGMENT_ID, in_numRows),
   m_columnData(in_columnData),
   m_rows(in_rows),
   m_startRow(in_startRow)
   {
       // Do nothing.
   }
   ```

2. Generate a unique "strategy" ID for the new class and pass this to `AbstractColumnSegment`'s constructor (note that ID's less than `AbstractColumnSegment::STARTCUSTOM_ID` are reserved by the SDK). For example, `XMColumnSegment` defines this ID as a static member called `XM_COLUMNSEGMENT_ID` and then passes it to `AbstractColumnSegment` constructor in the member initialization list. This is used by the Bulk Converter to determine which type of concrete `Column` Segment has been passed to it.

3. Modify your implementation of your IResult's `BulkFetch` method (or `DoBulkFetch` if subclassing from `DSISimpleResultSet`) to perform or delegate the bulk fetch process. Since the SQLite sample connector demonstrates the use of different Column Segment types based on that specified in its connector settings, it uses a switch/case statement in `XMTableLight::DoBulkFetch` to check which Column Segment type was specified and delegates accordingly. For example, if `XMStringColumnSegment::XM_COLUMNSEGMENT_ID` was specified in the connector's settings (stored in the class's `m_columnSegmentId` member), it delegates the bulk fetch to an XMTableColumnDataBase object:

   ```
   switch (m_columnSegmentId)
   {
   ```

```
....
case XMStringColumnSegment::XM_COLUMNSEGMENT_ID:
{
    column.Process(processor, m_rows, firstRow, rowsToReturn,
    rightTrimmer);
    break;
}
...
}
```

Note:

Connectors requiring their own custom Column Segment implementation will always use their implementation rather than perform the check, as was illustrated above, to determine the type. However a connector could implement different column segment types depending on the metadata of the column or mix the use of SDK Column Segments for some columns with custom Column Segments for other columns.

4. Instantiate your custom Column Segment, and pass it to the Bulk Processor's Process method to perform the bulk fetch. For example, the `XMTableColumnDataBase` object's `Process` method instantiates an `XMColumnSegment` and passes it directly to the Bulk Processor's `Process` method:

```
template<typename T> void XMTableColumnData<T>::Process(
  Simba::DSI::IBulkProcessor& in_bulkProcessor,
  const std::vector<RowIdentifier>& in_rows,
  simba_unsigned_native in_startRow,
  simba_unsigned_native in_numRows,
  RightTrimmer* in_rightTrimmer) const
{
UNUSED(in_rightTrimmer);
  in_bulkProcessor.Process(XMColumnSegment<T>(m_dataColumn, in_rows, in_startRow, in_
numRows));
}
```

5. Implement a custom `ISqlToCBulkConverter` class which can perform a conversion using your custom Column Segment class. For example, a connector might implement a `XMSqlToCBulkConverter` class to handle conversions for XM's underlying database. Its `Convert` method iterates through each row to fetch, invoking `XMColumnSegment::GetData` to return the address and data size for a cell from the underlying data source as a pair. It then uses that information to perform the fetch and conversion of data for the cell by invoking the conversion functor operator():

```
for (
```

simba_unsigned_native row = columnSegment.m_startRow, endRow = row + columnSegment.GetNumberRows();

row < endRow;

++row, ++currentRow1Based, targetPtr += in_toDataOffset, targetLenPtr = reinterpret_cast<simba_signed_native*>(reinterpret_cast<simba_byte*>(targetLenPtr)+in_toLengthOffset))

{

const std::pair<const void*, simba_uint32> data(columnSegment.GetData(rowIDs[row % numRowIDs]));

if (!data.first)

{

*targetLenPtr = CvtLength::MakeNull();

}

else

{

*targetLenPtr = in_toDataLength;

(*this)(

data.first,

data.second,

targetPtr,

*targetLenPtr,

in_listener);

}

}

6. Create a mapping between your custom converter and all SQL types that it should convert. In the SQLitesample connector, the `XMSqlToCBulkConverterWrapperMap` defines the mapping between the SQL types the connector supports and the template class (implementing `ISqlToCBulkConverterWrapper`) with which to wrap the functors for that destination SQL type.

7. Create a custom class template with the same interface as `DefaultSqlToCBulkBuilderFuncGenerator` (provided by the SDK). For convenience, you can reuse some definitions from `DSISqlToCBulkBuilderFuncGenerator.h`. The struct must have a static `GetBuilder` method which takes in a reference to an `IConnection` and returns a new `SqlToCBulkBuilderFunction`. `SqlToCBulkBuilderFunction` is a pointer to a factory function used by a Bulk Converter factory to create the converter. The following sample shows one way of implementing this functionality:

   template <TDWType SqlType, TDWType SqlCType> struct XMSqlToCBulkBuilderFuncGenerator {static SqlToCBulkBuilderFunction GetBuilder(Simba::DSI::IConnection& in_connection){

   return Simba::DSI::Impl::SqlToCBulkBuilderFuncGenerator<

   Simba::DSI::Impl::SENSqlToCConversionSupport<SqlType, SqlCType>::IsSupported,

```
SqlType,
SqlCType,
Simba::DSI::Impl::DSISqlToCBulkConverterFunctorMap,
XMSqlToCBulkConverterWrapperMap,
CharToCharIdentEncCvtFunctor,
CharToFromWCharCvtFunctor>::GetBuilder(in_connection);
}
};
```

This method delegates the creation to the SDK's SqlToCBulkBuilderFuncGenerator::GetBuilder method, but passes the XMSqlToCBulkConverterWrapperMap type as a template parameter.

8. Override or modify the `GetSqlToCBulkConverterFactory` method in your `DSIConnection`-derived class so that it instantiates a new `DSISqlToCBulkConverterFactory` using your custom `SqlToCBulkBuilderFunction` function. The following code snippet shows the implementation of `XMConnection::GetSqlToCBulkConverterFactory` which first checks if a factory has been created, and if not, checks to see if the SQLite Column Segment ID was specified. If it was specified, then a new `DSISqlToCBulkConverterFactory` is created using XM's custom `XMSqlToCBulkBuilderFuncGenerator` as the template type:

```
const ISqlToCBulkConverterFactory& XMConnection::GetSqlToCBulkConverterFactory(
{
  if (m_sqlToCBulkConverterFactory.IsNull())
  {
    if (XMStringColumnSegment::XM_COLUMNSEGMENT_ID == m_XMSettings.m_columnSegmentId)
    {
      m_sqlToCBulkConverterFactory.Attach(
        new DSISqlToCBulkConverterFactory<XMSqlToCBulkBuilderFuncGenerator>(*this));
    }
    else
    {
     DSIConnection::GetSqlToCBulkConverterFactory();
    }
  }
 return *m_sqlToCBulkConverterFactory;
}
```

Creating a Custom Conversion Functor

A conversion functor is an object that defines the `operator()` method for an `ISqlToCBulkConverter` implementation, to convert a specific SQL type to a specific C type. More specifically, it performs the conversion and copying of a single data cell of a specific data type, from the source to the target locations passed to it by the `SqlToCBulkConverterWrapper` that contains the converter and invokes its `operator()`.

The SDK defines a generic conversion functor template class called `SqlToCFunctor` along with templated operator() methods for all SQL-to-C data type conversions supported by the SDK, however developers are free to extend or create customized functors (e.g. to convert a special data type in your connector).

Note:

Extending or customizing functors can be done independently of creating a custom Column Segment. A custom Column segment doesn't require a custom conversion functor, and a custom conversion functor does't require a custom Column segment.

The following outlines the steps required to add an `operator()` method:

1. Define a new functor class, similar to `SqlToCFunctor`. This class can optionally be a template class which takes in template parameters specifying the SQL and C types to convert from and to. This class must also define `operator()` with the same parameters that `SqlToCFunctor`'s `operator()` takes in:

   ```
   void operator()(

   const void* in_source,

   simba_signed_native in_sourceLength,

   void* in_target,

   simba_signed_native& io_targetLength,

   IConversionListener& in_listener);
   ```

2. Create a new mapping between the converter and functor by defining a templated structure which maps the SQL and C types for which the functor is to convert. The SDK defines the following default mapping for the basic SQL-to-C conversions:

   ```
   template <TDWType SqlType, TDWType SqlCType>

   struct DSISqlToCBulkConverterFunctorMap

   {

      typedef SqlToCFunctor<SqlType, SqlCType> Type;

   };
   ```

   A connector can then extend this map as required. For example, a connector could define functor classes called `CustomCharConversionFunctor` and `CustomIntToStringConversionFunctor` to perform custom conversions of characters and integers to strings respectively, after which, the following maps could be defined:

   ```
   template <TDWType SqlType, TDWType SqlCType>

   struct CustomSqlToCBulkConverterFunctorMap
   ```

```
{
   typedef DSISqlToCBulkConverterFunctorMap<SqlType, SqlCType> Type;
};
template <TDWType SqlCType>
struct CustomSqlToCBulkConverterFunctorMap<TDW_SQL_CHAR, SqlCType>
{
   typedef CustomCharConversionFunctor<SqlCType> Type;
};
template <>
struct CustomSqlToCBulkConverterFunctorMap<TDW_SQL_SINTEGER, TDW_C_CHAR>
{
   typedef CustomIntToStringConversionFunctor Type;
};
```

3. Specify the mapping in your `SqlToCBulkBuilderFuncGenerator`'s `GetBuilder()` method. The following code snippet shows the SQLitesample's custom `GetBuilder()` method:

```
struct XMSqlToCBulkBuilderFuncGenerator
{
  static SqlToCBulkBuilderFunction GetBuilder(Simba::DSI::IConnection& in_connection)
  {
    return Simba::DSI::Impl::SqlToCBulkBuilderFuncGenerator<
    Simba::DSI::Impl::SENSqlToCConversionSupport<SqlType, SqlCType>::IsSupported,
    SqlType,
    SqlCType,
    Simba::DSI::Impl::DSISqlToCBulkConverterFunctorMap,
    XMSqlToCBulkConverterWrapperMap,
    CharToCharIdentEncCvtFunctor,
    CharToFromWCharCvtFunctor>::GetBuilder(in_connection);
  }
 };
```

   The parameter: `Simba::DSI::Impl::DSISqlToCBulkConverterFunctorMap` can replaced with the mapping created in the previous step, for example, `CustomSqlToCBulkConverterFunctorMap`.

For more information on optimizing data retrieval, see http://www.simba.com/blog/optimization-of-odbc-data-retrieval-with-the-simbaengine-sdk/

Parsing ODBC and JDBC Escape Sequences

Many SQL-enabled data stores represent data and implement SQL in slightly different ways. To allow applications to handle these differences transparently, the ODBC and JDBC standards specifies a set of escape sequences to represent functionality such as date, time, scalar functions, and procedure calls. ODBC and JDBC connectors must translate these escape sequences into a format that their data store supports.

The Simba SDK includes the MiniParser feature to help developers parse SQL commands for escape sequences, then replace them with the command format understood by their data store. SQL commands can contain multiple escape sequences with multiple parameters, and escape sequences themselves can be nested. The MiniParser implements all of the recursive processing and the creation of complex regular expressions required to support escape sequences. Using the Simba SDK, it is easy for your connector to translate SQL commands containing complex, nested escape sequences into a format that your data store understands.

ODBC and JDBC Escape Sequences

Escape sequences are grouped into types, making them easier to parse and process. Notice they are all enclosed in curly braces (`{ }`). For example, some common escape sequences are shown below:

Escape sequence type	Format	Example
date	{d 'value'}	{d '2001-01-01'}
scalar function	{fn scalar-function}	{ fn DAYOFWEEK(DATE '2000-01-01') }
procedure call	{[?=]call procedure-name[([parameter][, [parameter]]...)]}	{?=call LENGTH ('hello world')}

For information about the complete set of ODBC escape sequences, see "*ODBC Escape Sequences*" in the ODBC Programmer's Reference: https://msdn.microsoft.com/en-us/library/ms711838(v=vs.85).aspx. For information about JDBC escape sequences, see http://docs.oracle.com/cd/E13222_01/wls/docs91/jdbc_drivers/sqlescape.html.

Note:

The Simba SDK handles all escape sequences in the ODBC and JDBC specification.

Converting Simple Escape Sequences

Connectors must locate escape sequences and convert them to commands that are understood by their data source.

Simple Example: Dates

Consider a SQL command that contains an escape sequence of type date:

SELECT OrderNum, OrderDate FROM Orders WHERE OrderDate = {d '2015-08-12'}

A PostgreSQL connector converts the command as:

SELECT OrderNum, OrderDate FROM Orders WHERE OrderDate = DATE '2015-08-12'.

A Microsoft SQL connector converts the command as:

SELECT OrderNum, OrderDate FROM Orders WHERE OrderDate ='08-12-2015'.

Converting Complex Escape Sequences

Escape sequences can be nested, requiring recursive programming to replace them correctly.

Complex Example: Nested Escape Sequences

Given an escape sequence with the following format:

{fn EXTRACT(YEAR FROM {ts '2001-02-03 16:17:18.987654'}) }

A PostgreSQL connector converts the command as:

EXTRACT(YEAR FROM TIMESTAMP '2001-02-03 16:17:18.987654')

Non-Escaped Scalar Functions

Some applications use ODBC scalar functions in a SQL command without enclosing the function in an escape clause. For example, an application might use `CONVERT(sqltype,value)` instead of `{fn CONVERT(value, odbctype)}`. The miniParser handles the CONVERT scalar function in non-escaped form. Currently, other non-escaped scalar functions are not handled.

MiniParser Architecture

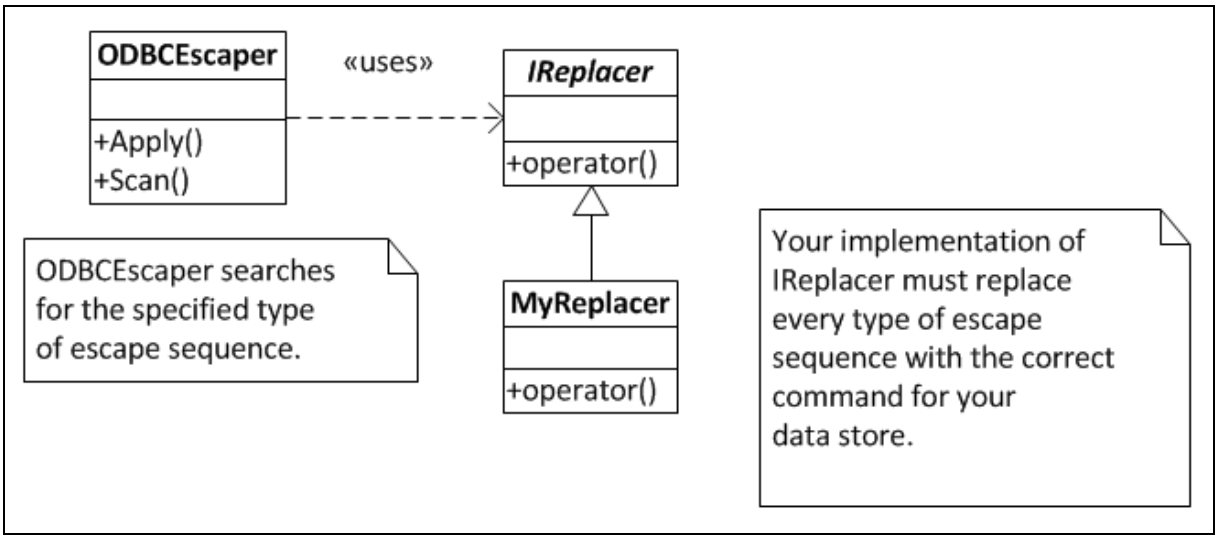
The miniParser is included in the `Support` package of the Simba SDK. It is composed of two main classes:

ODBC Architecture:

- **`ODBCEscaper`**: searches the SQL command for ODBC escape sequences and parameters. To use this class, pass an `IReplacer` implementation and the SQL command to `ODBCEscaper.Apply()`.

- **`IReplacer`**: converts each type of escape sequence to the format required for a particular data store. Override this class to provide your own implementation.

This architecture is shown in the figure below:



JDBC Architecture:

- **`JDBCEscaper`**: searches the SQL command for JDBC escape sequences and parameters. To use this class, pass an `IReplacer` implementation and the SQL command to `JDBCEscaper.Apply()`.
- **`IReplacer`**: converts each type of escape sequence to the format required for a particular data store. Override this class to provide your own implementation.

Error Handling

This section explains how `ODBCEscaper` handles errors and malformed SQL statements.

Text in unsupported locations is discarded

If a SQL statement is incorrectly formed and contains text in unsupported locations, `ODBCEscaper` will discard the text. For example, the escape sequence `{fn ABS(myNum)  bad string}` is incorrectly formed, as no text is allowed after the function name. In this case, `ODBCEscaper` will discard the text `bad string`.

Incorrectly formatted escape sequences are not sent to IReplacer

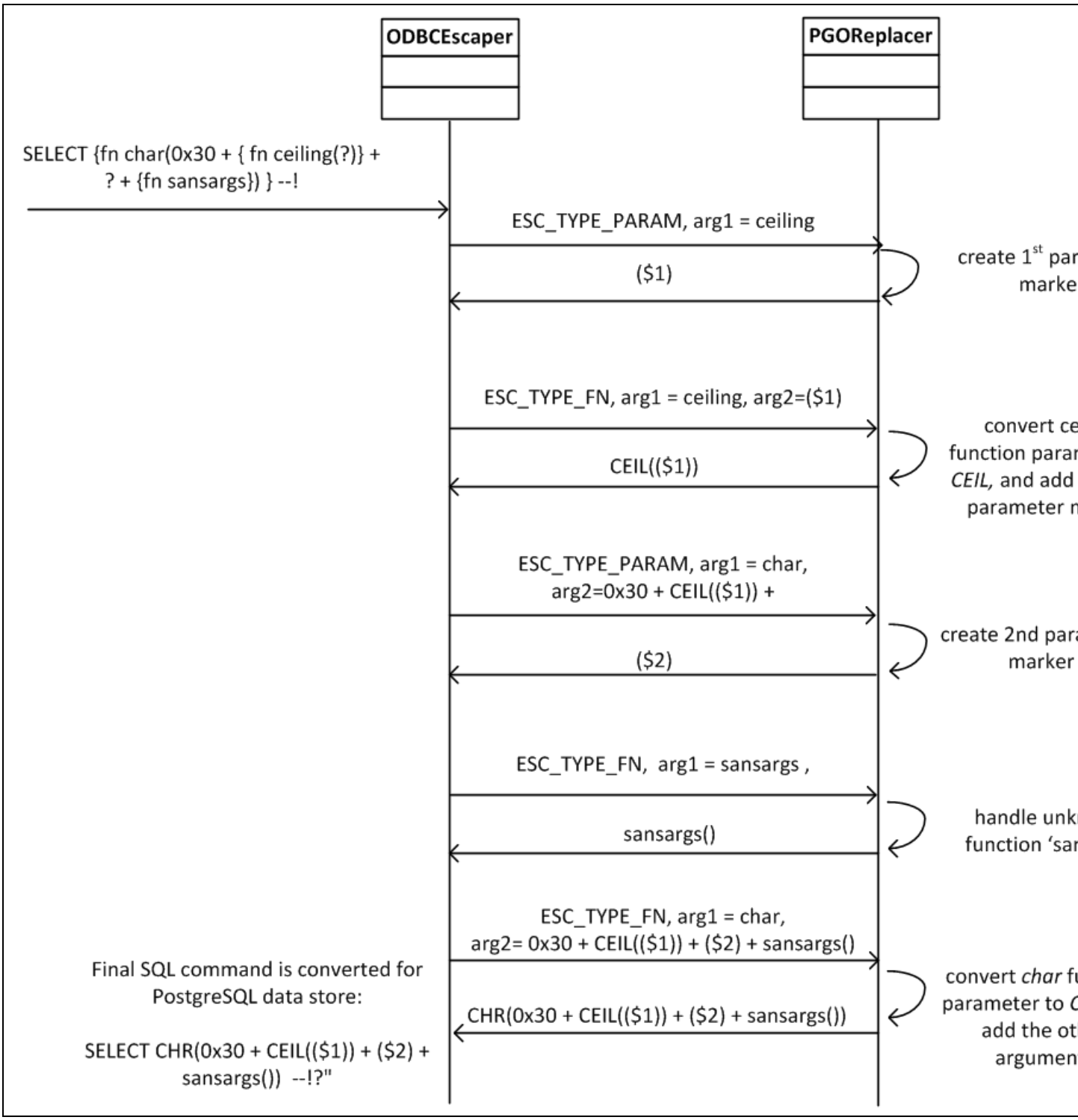
If an escape sequence is incorrectly formatted, `ODBCEscaper` will not pass it through to `IReplacer`, and will leave it unchanged. For example, `{D 2001-1-1}` is incorrectly formatted because it does not contain quotation marks (`''`). The incorrect escape sequence is simply included in the final SQL command. This allows the data store to handle the incorrect command sequence with the appropriate error.

Example Workflow

The following diagram shows how the `ODBCEscaper` and a sample `IReplacer` implementation, `PGOReplacer`, work together to convert a SQL statement containing parameters and escape sequences into a SQL statement for a PostgreSQL data store.

Note:

The work flow is the same for JDBCEscaper.



1. The connector calls `ODBCEscaper.Apply()`, passing in the SQL command `SELECT {fn char (0x30 + { fn ceiling(?)} + ? + {fn sansargs}) } --!?`. This command contains nested functions, a custom (non-ODBC) function, and a comment string.

2. `ODBCEscaper` starts with the inner most escape sequence, `{fn ceiling(?)}`. First, it tells `PGOReplacer` to create the parameter marker.
3. `PGOReplacer` creates and returns the first parameter marker in the format that the PostgreSQL data store understands.
4. `ODBCEscaper` then tells `PGOReplacer` to handle `{fn ceiling(?)}`, passing in the first converted parameter, `($1)`.
5. `PGOReplacer` converts the ceiling function to CEIL as required by the PostgreSQL data store, and uses the parameter marker `($1)` in the function.
6. This process repeats until `IReplacer` converts all the escape sequences and parameter markers. Then `ODBCEscaper` reassembles the SQL command, including the comment and the string.

The original SQL command is now converted into a SQL command that PostgreSQL data store can understand.

Example Implementation

This section shows an example implementation of `IReplacer` for a custom ODBC or JDBC connector. The following steps are required:

- "Step 1: Implement Your Custom IReplacer" below
- "Step 2: Create an Instance of ODBCEscaper" on page 75
- "Step 3: Ensure Additional Requirements are Met" on page 75

Step 1: Implement Your Custom IReplacer

Implement your `IReplacer` to convert ODBC standard escape sequences to the commands that your data store understands.

Note:

- For Java, use `JDBCEscaper` instead of `ODBCEscaper`. All other methods and techniques shown in this section are the same.
- In this example, `IReplacer` handles a subset of the possible ODBC escape sequences. Typically, your custom ODBC connector implements the complete set of ODBC escape sequences described in https://docs.microsoft.com/en-us/sql/odbc/reference/appendixes/odbc-escape-sequences.

Example

```
#include <Simba.h>

#include <ODBCEscaper.h>

static char const* keyword[] = {

        "DATE ", "ESCAPE ", "TIME "

};

class MyReplacer : public IReplacer
```

```
{
        MyReplacer()
        {
                m_numParams = 0;
        }
        simba_wstring operator()(ODBCEscaper::ESC_TYPE in_etype, std::vector<simba_
        wstring>& args)
        {
                switch (in_etype)
                {
                // Date, Time, and Timestamp Escape Sequences
                case ODBCEscaper::ESC_TYPE_DATE:
                case ODBCEscaper::ESC_TYPE_ESCAPE:
                case ODBCEscaper::ESC_TYPE_TIME:
                {
                        return simba_wstring(keyword[in_etype - ODBCEscaper::ESC_
                TYPE_DATE]) + args[0];
                }
                break;
                // Here replace ? with ($1)....
                case ODBCEscaper::ESC_TYPE_PARAM:
                {
                        char buf[99];
                        sprintf(buf, "($%d)", ++m_numParams);
                        // implicit conversion to simba_wstring
                        return buf;
                }
                break;
                //Scalar Functions Escape sequences.
                case ODBCEscaper::ESC_TYPE_FN:
                {
```

```
            if (args[0].IsEqual("CEILING", false))
            {
            args[0] = "CEIL";
            }
            else if (args[0].IsEqual("CHAR", false))
            {
            args[0] = "CHR";
            }
            else if (args[0].IsEqual("POWER", false))
            {
            args[0] = "POW";
            }
            if ((args[0].IsEqual("CONVERT", false)) && (3 == args.size()))
            {
            args[0] = "CAST";
            }
            return args[0] + "(" + simba_wstring::Join(args.begin() + 1,
            args.end(), ", ") + ")";
}
break;
// Handling the non-escaped scalar functions: Note different argument order.
case ODBCEscaper::ESC_TYPE_FUNC:
{
            if ((args[0].IsEqual("CONVERT", false)) && (3 == args.size()))
            {
                        return "CAST_RAW(" + args[2] + " AS " + args[1] + ")";
                        }
            }
            break;
            // unimplemented Escape Types.
            default:
```

```
                {
                        return simba_wstring("TODO: ") +
                ODBCEscaper::type_name[in_etype];
        }
        break;
        }
    }
}
private:
    int m_numParams;
};
```

Step 2: Create an Instance of ODBCEscaper

`ODBCEscaper` or `JDBCEscaper` handles the parsing of the SQL command, identifying parameter markers and escape sequences while passing over the contents of strings, identifiers and comments. It passes each parameter marker and escape sequence to `IReplacer`, along with the type and argument information. `IReplacer` returns the converted command.

Parsing is done from left to right, and in the case of nested escape sequences, from the inner to the outer brackets. When the parsing and replacements are finished, `ODBCEscaper` or and `JDBCEscaper` reassemble the SQL command, adding back any strings or comments.

Create an instance of `ODBCEscaper`, then call `ODBCEscaper.Apply()`, passing a instance of your custom `IReplacer` and the SQL command to parse and convert. Because `IReplacer` maintains state for the duration of a SQL command, you must create a new `IReplacer` for each SQL command that you want to parse.

Example:

```
ODBCEscaper esc;
MyReplacer replacer;
simba_wstring newSQLstr;
// newSQLstr will contain the converted SQL command
simba_wstring newSQLstr = esc.Apply(replacer, "SELECT {fn LOG( {fn LOG10({fn POWER(10,2)})})}");
```

Step 3: Ensure Additional Requirements are Met

This section contains additional information and requirements for implementing your `IReplacer`.

Return commands that are not ODBC or JDBC compliant

If `IReplacer` encounters a command escape sequence that is not part of the ODBC or JDBC specification, it should return the command back to `ODBCReplacer` without modification. This is

illustrated in the "Workflow" section in "Parsing ODBC and JDBC Escape Sequences" on page 68, as the parameter `sansargs` is not ODBC compliant.

Maintain a parameter count

Your `IReplacer` implementation must keep track of the number of parameter markers it returns so that it can increment them correctly. For example, “@1”, “@2”, “@3”, or ($1), ($2), ($3).

Reject unknown input

If your `IReplacer` implementation receives input that it does not know how to handle, it must throw an exception or return the string in curly brackets (`{ }`).

Important:

For security reasons, an `IReplacer` must never return a string that forces a syntax error.

Return an expression in parenthesis or surrounded by spaces

Where possible, the commands or expressions that your `IReplacer` returns should be surrounded by parentheses (`()`) or spaces ( ). This allows the ODBCEscaper to correctly reassemble the SQL command. The `IReplacer` sample surrounds the commands and parameters with parentheses. For example, when returning the value `['4:05'::TIME]`, format the value in one of the following ways:

- `[ '4:05'::TIME ]` // notice the spaces
- Or, `[('4:05'::TIME)]` // notice the parenthesis

Ensure correct syntax

In order for `ODBCReplacer` to correctly parse and reassemble the SQL statement, the `IReplacer` implementation must always return parameters and converted escape sequences that contain correct syntax.

> **Important:**
> **Important:**
>
> `IReplacer` must not return an odd number of quotes, an unterminated comment, or mismatched parentheses.

Related Topics

"Non-Escaped Scalar Functions" on page 69

"Error Handling" on page 70

Errors, Exceptions, and Warnings

This section explains how to implement the classes that handle errors, exceptions, and warnings. It also explain how to use and localize the files that contain error messages. Error message files are available in several different languages.

Handling Errors and Exceptions

ODBC, JDBC, and ADO.NET require that connector provide standard error codes so that applications have a standard way of dealing with error conditions. Data stores can also provide their own custom error codes. This section explains what your custom connector should do when it encounters an error or an exception.

Using the ErrorException Class

When your DSII detects an error condition, it should throw an exception of type `ErrorException`. This class has the following signature:

ErrorException(

DiagState in_stateKey,

simba_int32 in_componentId,

const simba_wstring& in_msgKey,

simba_signed_native in_rowNum = NO_ROW_NUMBER,

simba_int32 in_colNum = NO_COLUMN_NUMBER);

The parameters for this method are described below:

- **in_componentId**

  The component id is used to determine which component threw the exception and where the message should be loaded from. The list of reserved component Ids (1-10) and their names can be found in `SimbaErrorCodes.h`. It is suggested that any custom component Id you define for your DSII start counting from 100.

- **in_msgKey**

  The `in_msgKey` argument is a string shortcut to indicate which message to load from the standard error message file or your own custom message source. For information about error message files, see "Localizing Messages" on page 83.

- **in_stateKey**

  The in_stateKey argument is used to control which SQLSTATE code should be associated with the error returned by ODBC. SQLSTATE is a 5-character sequence defined by SQL standards that is used to return a standard error code. The most common state to throw is DIAG_GENERAL_ERROR. A full list of available DiagState keys can be found in `DiagState.h`.

Exception Macros in the Sample Connectors

The Quickstart sample connector provides sample macros that you can adapt to throw your own exceptions. These macros are defined in `Quickstart.h`. For information on using Quickstart, see the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

Example: Using Quickstart's Exception Macro

In the sample Quickstart connector, the following macro is used to throw an exception if the required DBF setting is missing:

```
QSTHROW(DIAG_INVALID_AUTH_SPEC, L"QSDbfNotFound");
```

This throws an `ErrorException` with a `DiagState` of DIAG_INVALID_AUTH_SPEC and the `QSDbfNotFound` message key. The macro automatically includes the Quickstart component Id.

Some messages are also parameterized, and there are sample macros to assist in constructing the vector of parameters before throwing the exception.

Example: Throwing an Exception With Parameters

This example throws an `ErrorException` with a DiagState of DIAG_GENERAL_ERROR and the `QSInvalidCatalog` message key.

```
QSTHROWGEN1(L"QSInvalidCatalog", in_schemaName);
```

`in_schemaName` is a `simba_wstring` parameter that is added to a vector and passed to a constructor for `ErrorException`, which accepts a parameter vector. The message source will use the parameter vector to do string substitution on special markers in the message string.

Using or Building a Message Source

All exceptions and warnings in your custom connector are looked up by their message key using an `IMessageSource` constructed by your custom connector. An implementation of this class, called `DSIMessageSource`, is provided to handle looking up any message key generated by SDK components. This class looks up the messages in the error messages files. The error message files are located in the directory `[INSTALL_DIRECTORY]\DataAccessComponents\ErrorMessages`. The connector determines the location of this file by looking up the `ErrorMessagesPath` value in the registry at **HKLM\Software\<OEM NAME>\Driver**, or inside the configuration file on Linux, Unix, or macOS platforms.

In order to provide messages of your own, you must register an error messages file with `DSIMessageSource` or construct your own `MyDSIIMessageSource` class deriving from `IMessageSource`. If you use `DSIMessageSource`, you will only be responsible for providing an XML message file for all of the messages your DSII uses. If you derive from `IMessageSource`, you will be responsible for looking up any message key generated by either the SDK or your DSII.

All of the sample connectors register an additional message file with the default `DSIMessageSource`, and it is recommended that your DSII do the same unless there is good reason to do otherwise. The error messages XML files are placed in directories named after the locale that the message files are associated with, for example, `[INSTALL_DIRECTORY]\DataAccessComponents\ErrorMessages\en-US`.

For information about error message files, see "Including Error Message Files" on page 80.

Custom SQL States

SQLSTATE is a 5-character sequence defined by SQL standards. It provides detailed information about the cause of a warning or error. The Simba SDK attempts to return SQL states, or equivalent, that accurately follow the specifications of ODBC, JDBC, and ADO.NET. However, in some cases your custom connector may need to return a different SQL state than what is used by the SDK. In those cases, your DSII will return a custom SQL state as described in the this section:

ODBC

Exceptions are implemented in the `ErrorException` base class. The predefined SQL states are mapped to `DiagStates`, and there are constructors that take a `DiagState` along with other information for this purpose. When using custom SQL states, use the constructors that take a `simba_string` for the SQL state to provide any 5 character SQL state.

Likewise, warnings with custom SQL states will post warnings to the `IWarningListener` using the `simba_string` constructor instead of the `DiagState` constructor.

JDBC

Exceptions are implemented in the `DSIException` base class. The predefined SQL states are mapped to `ExceptionID`, and there are constructors that take an `ExceptionID` along with other information for this purpose. When using custom SQL states, use the constructors that take a `String` for the SQL state to provide any 5 character SQL state.

Likewise, warnings with custom SQL states will post warnings to the `IWarningListener` using a `Warning` constructed with the `String` constructor instead of the `WarningCode` constructor.

ADO.NET

Exceptions are implemented in the `DSIException` base class. Note that SQL states are not directly supported by the ADO.NET API. Instead, the custom SQL state is prepended to the exception error message. The predefined SQL states are mapped to `ErrorCode`, and there are constructors that take an `ErrorCode` along with other information for this purpose. When using custom SQL states, use the constructors that take a string for the SQL state to provide any 5 character SQL state.

Likewise, warnings with custom SQL states will post warnings to the `IWarningListener` using the string constructor instead of the `WarningCode` constructor.

OLE DB

SQL states, custom or not, are exposed using the custom error object through the `ISqlErrorInfo` interface. For details on the `ISqlErrorInfo` interface, see http://msdn.microsoft.com/en-us/library/windows/desktop/ms711569%28v=vs.85%29.aspx. Also, refer to the topic *How a Provider Returns an OLE DB Error Object in MSDN* at http://msdn.microsoft.com/en-us/library/windows/desktop/ms723101%28v=vs.85%29.aspx.

Related Topics

"Posting Warning Messages" on the next page

"Including Error Message Files" on the next page

"Localizing Messages" on page 83

Posting Warning Messages

The Simba SDK supports warning messages in a similar way as it supports error messages.

Using the IWarningListener interface

You can post warnings to an `IWarningListener` interface. The `DataStoreInterface` core classes, `DSIEnvironment`, `DSIConnection` and `DSIStatement`, each have an associated `IWarningListener`. Your custom implementation of these classes can access an `IWarningListener` through the parent `GetWarningListener()` method.

For the complete list of warnings that can be posted to an `IWarningListener`, see the file `DataAccessComponents\Include\Support\DiagState.h` in your Simba SDK installation directory.

Similar to `ErrorException`, `IWarningListener` uses the error messages files associated with the `DSIMessageSource` to retrieve the warning messages corresponding to the error or warning code. For more information on this functionality, see "Including Error Message Files" below.

Subscribing to an IWarningListener

The SDK controls most of the classes that warning listeners can be registered with. This means you don't have to explicitly register them in your custom connector code. The one exception is the `ConnectionSetting` object - you must register the warning listener with this class after construction. See the example in "Handling Connections" on page 41.

Posting Warnings to an IWarningListener

Use `GetWarningListener()->PostWarning()` to post a warning to the warning listener.

Example: ConnectionSetting object Posting Warnings to an IWarningListener

```
// In CustomerDSIIEnvironment, CustomerDSIIConnection and
// CustomerDSIIStatement, use the parent GetWarningListener()
// function to retrieve the IWarningListener
this->GetWarningListener()->PostWarning(
Diagnostics::OPT_VAL_CHANGED,
ComponentKey,
L”WarningMessageKey”);
```

Related Topics

"Posting Warning Messages" above

"Including Error Message Files" below

"Localizing Messages" on page 83

Including Error Message Files

This section describes the error message files used by the SDK for ODBC and JDBC connectors.

Error Messages in ODBC

The ODBC error messages are defined in `.xml` files. The table below describes each file, and explains which error message files must be included when you distribute your connector.

Error Message File Name	Description	Do I Need to Ship this File?
`ODBCMessages.xml`	Contains the error messages for the ODBC, DSI, and Support components.	Yes, always with your connector. If you distribute SimbaClient for ODBC, you will also need to include this file.
`ClientMessages.xml`	Contains the error messages for SimbaClient for ODBC.	Only if you are distributing SimbaClient for ODBC.
`CSCommonMessages.xml`	Contains the error messages for the Client/Server protocol components.	Only if you have built your connector as a server. If you distribute SimbaClient for ODBC, you will also need to include this file.
`ServerMessages.xml`	Contains the error messages for SimbaServer.	Only if you have built your connector as a server.
`CLIDSIMessages.xml`	Contains the error messages for the CLIDSI component.	Only if your connector uses the CLIDSI component.
`JNIDSIMessages.xml`	Contains the error messages for the JNIDSI component.	Only if your connector uses the JNIDSI component.

Organizing your ODBC Error Message Files

By default the SDK uses the English - United States (en-US) locale. You can add support for additional locales by organizing your additional language files in one of the following ways:

Subdirectory organization

You can store each locale's message files in a subdirectory, where the subdirectory is named using the locale code.

Example: Subdirectory organization of message files

...\ErrorMessages\en-US\ODBCMessages.xml

...\ErrorMessages\fr-CA\ODBCMessages.xml

...\ErrorMessages\ja-JA\ODBCMessages.xml

Single directory organization

You can store all message files for every locale in a single folder. The name of each locale is added as a suffix in the file names.

Example: Single directory organization of message files

...\ErrorMessages\ODBCMessages_en-US.xml

...\ErrorMessages\fr-CA\ODBCMessages_fr-CA.xml

...\ErrorMessages\ja-JA\ODBCMessages_ja.xml

Error Messages in JDBC

The JDBC error messages are divided into several files. The table below describes each file, and explains which error message files must be included when you distribute your connector.

Error Message File Name	Description	Do I Need to Ship this File?
`JDBCMessages.properties`	Contains the error messages for the JDBC component.	Yes, always with your connector. If you distribute SimbaClient for JDBC, you will also need to include this file.
`DSIMessages.properties`	Contains the error messages for the DSI and Support components.	Yes, always with your connector. If you distribute SimbaClient for JDBC, you will also need to include this file.

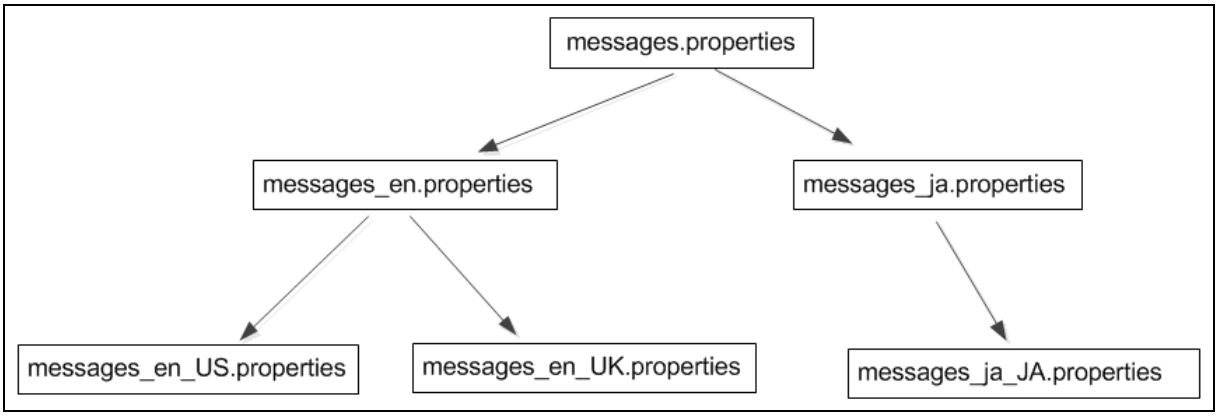
`CSMessages.properties`
`CommunicationsMessages.properties`
`Messages.properties`
	Contains the error messages for SimbaClient for JDBC and the Client/Server protocol components.	Only if you are distributing SimbaClient for JDBC.

Organizing your JDBC Error Message Files

By default, the SDK uses the English - United States (en-US) locale. You can add support for additional locales using Java Resource Bundles.

The common convention for localization with resource bundles is to organize the error message files in a hierarchy. This ensures that messages from a parent message file will be used, even if a locale is not supported.

For example, the structure for message files could be organized in the following hierarchy. In this example, is the base file name is *messages*:



Note:

Each message file must be registered separately with `DSIMessageSource`.

Related Topics

"Handling Errors and Exceptions" on page 77

"Posting Warning Messages" on page 80

"Localizing Messages" below

Localizing Messages

Simba SDK includes sample string resources for the warning and error messages that it may generate. These strings are provide in English, as well as other languages including German, French, Spanish, and Japanese. These files are intended as a starting point to aid you in the localization process. You can modify the localized strings that are provided, provide your own connector-specific messages, and add support for additional languages.

Note:

The files provided for languages other than English are not complete. Some of these strings are still in English and require further translation.

Customers can configure the locale, or language, of the messages that the connector uses. Configuration can be done connector-wide so that all connections use the same locale for their messages, or per-connection so each connection uses a different locale.

Configuring the Connector Locale

Customers can configure the locale of connector for all connections (connector-wide locale) or for individual connections. If the locale is not configured, the default locale of US English (en-US) is used for all messages.

Configuring the Connector-Wide Locale

A single locale is specified for the connector, and all connections use the same language for any messages.

To configure the connector-wide locale on Windows:

1. In the Windows registry, navigate to the following registry key:

- For 32-bit connectors on 32-bit machines or 64-bit connectors on 64-bit machines, navigate to **HKEY_LOCAL_MACHINE\SOFTWARE*<Company>**<ConnectorName>*\Driver**, where *<Company>* is your company name and *<ConnectorName>* is the name of your connector.
  For example, for the Simba Quickstart connector, navigate to **HKEY_LOCAL_MACHINE\SOFTWARE\Simba\Quickstart\Driver**.

- Or, for 32-bit connectors on 64-bit machines, navigate to **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node*<Company>**<ConnectorName>*\Driver**, where *<Company>* is your company name and *<ConnectorName>* is the name of your connector.
  For example, for the Simba Quickstart connector, navigate to **HKEY_LOCAL_MACHINE\SOFTWARE/Wow6432Node\Simba\Quickstart\Driver**.

2. In the **<Customer>/<ConnectorName>/Driver** section of the registry, add or modify the **DriverLocale** key to contain the desired locale code. For a list of locale codes, see "Locale Codes" on the next page.

To configure the connector-wide locale on Unix, Linux, and macOS:

1. Locate the `.ini` configuration file for the desired connector.
2. Modify the `DriverLocale` string to contain the desired locale code. For a list of locale codes, see "Locale Codes" on the next page.

Configuring Per-Connection Locale

A locale is configured for each connection, so each connection can use a different language for error messages. If the locale is not configured for a connection, then the connector-wide locale is used.

To configure the connection-wide locale on Windows:

1. In the Windows registry, navigate to the to the registry key for the DSN that is used for the connection:

- For 32-bit connectors on 32-bit machines or 64-bit connectors on 64-bit machines, navigate to **HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI*<Company>*DSII**, where *<Company>* is your company name.
  For example, for the Simba Quickstart connector, navigate to **HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\QuickstartDSII**.

- Or, for 32-bit connectors on 64-bit machines, navigate to **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432\ODBC\ODBC.INI*<Company>*DSII**, where *<Company>* is your company name.
  For example, for the Simba Quickstart connector, navigate to **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432\ODBC\ODBC.INI\QuickstartDSII**.

2. Modify the **Locale** key to contain the desired locale code. For a list of locale codes, see "Locale Codes" on the next page.

To configure the connector-wide locale on Unix, Linux, and macOS:

1. Locate the `.ini` configuration file for the desired connector.
2. Modify the `Locale` string to contain the desired locale code. For a list of locale codes, see "Locale Codes" below.

Locale Codes

Locales are specified using a two-letter language code in lower case and an optional two letter country code in upper case. If a country code is specified, it must be separated from the language code by a hyphen (-).

Examples:

- en-US (English – United States)
- fr-CA (French – Canada)
- it-IT (Italian – Italy)
- de-DE (German – Germany)
- es-ES (Spanish – Spain (Traditional))
- ja (Japanese)

The language code can be any language in the ISO 639-1 standard: http://www.loc.gov/standards/iso639-2/php/code_list.php. The country code can be any country in the ISO 3166-1 Alpha-2 standard: http://www.iso.org/iso/country_codes/iso-3166-1_decoding_table.htm.

Localizing Your Connector

The Simba SDK provides English strings for the error and warning messages that its components generate. These messages are contained XML files for ODBC and in a Java Resource Bundle for JDBC. When developing your own connector, you can create additional messages in English for any errors and warnings that are specific to your connector. To provide your connector-specific messages, create connector-specific XML files or Java a Resource Bundle containing your messages in the same format as the exiting Simba SDK message files. For information about error messages files, see "Including Error Message Files" on page 80.

`DSIMessageSource` automatically handles the loading and exposure of these messages to your connector. Your connector has to call `DSIMessageSource::RegisterMessages`, passing in the root name of the connector specific message file. The root name is the file name without an extension or locale code. For example, the root name for the QuickStart connector is `QSMessages`. A good place to call this method is in the constructor of the connector class that inherits from `DSIDriver`.

The connector can also implement its own message source by inheriting from `DSIMessageSource` and handling connector-specific messages, which may be in different format and location than those from the Simba SDK. For example, the messages may be stored in a database. The handling of SDK messages in this case can still be delegated to `DSIMessageSource`. Alternatively, `IMessageSource` can be implemented directly, but the implementation must handle both the connector specific messages and the Simba SDK messages. For more information on implementing error messages, see "Using or Building a Message Source" on page 78 in "Handling Errors and Exceptions" on page 77.

To support a locale for which the Simba SDK provides a translation when using the default `DSIMessageSource` class, translate the messages in your connector-specific message file and follow the naming convention described in the following subsections. To support a locale for which the SDK does not provide a translation, translate both the connector-specific and Simba SDK message files.

Additional Language Support

In addition to the languages that are shipped with the Simba SDK, translated messages strings for other languages are also available. For more information on obtaining these strings, contact Simba Technologies Inc.

Related Topics

"Localizing Messages" on page 83

"Posting Warning Messages" on page 80

"Including Error Message Files" on page 80

"Localizing Messages" on page 83

Multithreading

The Simba SDK typically handles all processing in a single thread, using the same thread as the application uses to make the ODBC or JDBC request. However, multiple threads may be started in the following cases:

- If the application creates a new thread for each ODBC or JDBC connection, each request is processed on its own thread. Processing is handled concurrently.
- In a client/server deployment, multiple clients can send a request to the same Simba Server. SimbaServer handles each request on its own thread.

In addition, the Simba SDK provides support for multithreading that you can use in your custom ODBC or JDBC connector.

Using the Thread Class (C++ only)

The `Thread` class provides the implementation for a thread. There are different options for using this class in your custom connector:

- You can subclass the `Thread` class and implement the `DoExecute()` interface.
- Or, you can call `StartDetachedThread()`, passing in a pointer to a function that will be executed when the thread is started.

Note:

There is no overall difference in functionality between these methods.

Using the ThreadPool Class

The `ThreadPool` class starts and manages the running threads. It implements the pool of threads, and is responsible for creating new threads and assigning tasks to them.

To implement a multi-threaded environment using the ThreadPool class:

1. To make a runnable task, subclass `ITask` and implement the `Run()` method.
2. Call the `PostTask()` method to add runnable tasks to a queue of unprocessed tasks on the `ThreadPool` class.

Note:

The maximum number of threads is specified by `m_maxThreads`.

Asynchronous ODBC Support

The Simba SDK enables your custom ODBC connector to support asynchronous ODBC. ODBC 3.8 supports asynchronous execution of ODBC connection functions, while ODBC 3.52 only supports asynchronous execution of statement functions. For more information about asynchronous ODBC support, see http://msdn.microsoft.com/en-us/library/ms713563%28v=vs.85%29.aspx

Simba SDK 9.3 and later releases supports the polling method for this asynchronous functionality. However, this support varies by platform as listed below.

Note:

Executing functions asynchronously using the polling method involves calling the same function is repeatedly until the function no longer returns SQL_STILL_EXECUTING. When repeatedly calling the function in such a loop, it's recommended that the same parameters be passed each time and that their values remain unchanged. This will prevent any unexpected errors from occurring.

Windows 7 +

- SQLBROWSECONNECT
- SQLCOLATTRIBUTE
- SQLCOLUMNPRIVILEGES
- SQLCOLUMNS
- SQLCONNECT
- SQLDESCRIBECOL
- SQLDESCRIBEPARAM
- SQLDISCONNECT
- SQLDRIVERCONNECT
- SQLENDTRAN
- SQLEXECDIRECT
- SQLEXECUTE
- SQLFETCHSCROLL
- SQLFETCH
- SQLFOREIGNKEYS
- SQLGETDATA
- SQLGETTYPEINFO
- SQLMORERESULTS
- SQLNUMPARAMS
- SQLNUMRESULTCOLS

- SQLPARAMDATA
- SQLPREPARE
- SQLPRIMARYKEYS
- SQLPROCEDURECOLUMNS
- SQLPROCEDURES
- SQLPUTDATA
- SQLSETPOS
- SQLSPECIALCOLUMNS
- SQLSTATISTICS
- SQLTABLEPRIVILEGES
- SQLTABLES

Non-Windows including iODBC, UnixODBC, SimbaDM

- SQLCOLUMNPRIVILEGES
- SQLCOLUMNS
- SQLEXECDIRECT
- SQLEXECUTE
- SQLFETCHSCROLL
- SQLFETCH
- SQLFOREIGNKEYS
- SQLGETTYPEINFO
- SQLPRIMARYKEYS
- SQLPROCEDURECOLUMNS
- SQLPROCEDURES
- SQLSPECIALCOLUMNS
- SQLSTATISTICS

- SQLTABLEPRIVILEGES
- SQLTABLES

Note:

- Asynchronous functionality at the connection level is not supported on non-Windows platforms.

Critical Section Locks

A critical section is a section of code that accesses a shared resource, where this resource must not be accessed at the same time as another thread. For example, only one thread at a time should write to a log file. If multiple threads write to a log file at the same time, the resulting text in the file could be an unpredictable mix up of text from each thread.

It is important to implement critical section locks when using either the Java or the C++ SDKs. If you are using the Java SDK, you can use standard Java classes to handle locking. If you are using the C++ SDK, you can use the classes provided by the Simba SDK.

Critical Section Locks in the C++ SDK

A critical sections of code should be specific using a `CriticalSection` object. A `CriticalSectionLock` object can then be used to lock this critical section to prevent concurrent access by another thread.

Tip:

Your implementation of `GetDriverLog`, for example `CustomerDSIIDriver::GetDriverLog`, should use a `CriticalSectionLock`.

To use critical sections and critical section locks:

1. Include the following files:

   #include “CriticalSection.h”

   #include “CriticalSectionLock.h”

2. Define a `CriticalSection` member variable. For example:

   ```
   Simba::Support::CriticalSection m_criticalSection;
   ```

3. For functions that use shared resources, use a `CriticalSectionLock` to lock the critical section. Add the following line of code to the start of the function:

   ```
   CriticalSectionLock lock(&m_criticalSection);
   ```

   The lock will be released once the function returns.

For more information on the `CriticalSection` and `CriticalSectionLock` classes, see the Simba SDK C++ API Reference.

Concurrency Support

Some ODBC functions can be run concurrently on statements that share the same connection, while other functions block.

For example, the ODBC catalog function `SQLTables` cannot be run concurrently. Suppose a thread is executing `SQLTables` on a statement, while another thread attempts to execute a function on another statement that shares the same connection. The second thread blocks until `SQLTables` on the first thread is finished.

This section explains the concurrency behaviour for the different ODBC functions, and explains how to change the behaviour from concurrent to blocking.

ODBC Functions that Support Concurrency

By default, the following ODBC functions support concurrency:

- `SQLPrepare`
- `SQLCloseCursor`
- `SQLFreeStmt`
- `SQLMoreResults`
- `SQLAllocHandle`
- `SQLFreeHandle`

These functions can be executed concurrently, even if the statements that are executing them share the same connection. For example, suppose a statement is executing a function on a connection. If you pass that connection handle to `SQLAllocHandle()`, the `SQLAllocHandle()` function is executed concurrently and does not block.

Similarly, suppose two statements, `Statement1` and `Statement2`, are using the same connection. `Statement1` is already executing. You can call `SQLFreeHandle()` on `Statement2` and it will not block.

Overriding the Default Behaviour

If you want these functions to block by default, you can change the default behaviour by setting a connector property.

To set ODBC functions to block:

- In your `IDriver` implementation, set the DSI_DRIVER_ALLOW_INCREASED_ODBC_STATEMENT_CONCURRENCY property to `false`:

```
SetProperty(DSI_DRIVER_ALLOW_INCREASED_ODBC_STATEMENT_CONCURRENCY,
AttributeData::MakeNewUInt32AttributeData(DSI_AIOSC_FALSE));
```

ODBC Functions that Do Not Support Concurrency

The following ODBC functions do not support concurrency:

- `SQLExecute`
- `SQLExecDirect`
- All catalog functions.

API Overview

This section introduces the functionality and workflows of the C++ DSI API and the DSI API Extensions, which are the main APIs that you use to build a custom connector. The Java APIs are similar.

DSI API

The DSI API exposes the classes needed to build your own Data Store Interface Implementation using C++. The C# and Java versions of these classes, the DotNet DSI API and the Java DSI API, provide similar functionality as the C++ classes.

The DSI API functionality is grouped into Core classes and Data Engine classes.

Core classes

The Core classes provide all of the essential functionality to establish and manage the connection to your data source:

Class	Description
`IDriver`	`IDriver` is a singleton instance constructed when the connector is first loaded. Its primary responsibility is to construct `IEnvironment` objects and manage any connector-wide properties. An abstract base class `DSIDriver` is provided to assist in some of these responsibilities, including initializing defaults and managing properties.
`IEnvironment`	`IEnvironment` objects correspond to the ODBC environment (ENV) handles allocated by `SQLAllocHandle`. Their primary responsibility is to construct `IConnection` objects and manage any environment properties. An abstract base class `DSIEnvironment` is provided.
`IConnection`	`IConnection` objects correspond to the ODBC connection (DBC) handles allocated by `SQLAllocHandle`. Their primary responsibility is to handle user authentication, construct `IStatement`objects, and manage any connection properties. An abstract base class `DSIConnection` is provided.
`IStatement`	`IStatement` objects correspond to the ODBC statement (STMT) handles allocated by `SQLAllocHandle`. Their primary responsibility is to construct `IDataEngine` objects and manage any statement properties. An abstract base class `DSIStatement` is provided.
`IMessageSource`	`IMessageSource` is responsible for loading error messages and warnings from your connector. An abstract implementation `DSIMessageSource` is provided to load messages generated by the SDK. For more information, see "*Using or building a message source*" in "Handling Errors and Exceptions" on page 77.

Class	Description
`ILogger`	`ILogger` is responsible for storing or printing log messages from your connector. Each of the `IDriver`, `IEnvironment`, `IConnection`, and `IStatement` classes has a `GetLog()` method which must return the most appropriate logger for that object. You may share loggers between all the objects or construct a different logger for each. The `DSIFileLogger` class is fully implemented to store the log messages to a text file, but you may change the behaviour in any way by extending the `ILogger` interface directly or by subclassing the partially implemented `DSILogger`class.

Data Engine classes

The Data Engine classes are the subset used to perform the data access functions against your data store:

Class	Description
`IDataEngine`	`IDataEngine` is responsible for constructing an `IQueryExecutor` when preparing queries or constructing an `IResult` for catalog function metadata. An abstract base class `DSIDataEngine` is provided to assist in implementing filters for the catalog function metadata.
`IQueryExecutor`	`IQueryExecutor` is responsible for executing a query and generating `IResults` objects.
`IResults`	An `IResults` object represents a collection of one or more `IResult` objects. `DSIResults` provides a basic implementation for accessing and managing a collection of `IResult` objects.
`IResult`	`IResult` is responsible for retrieving column data and maintaining a cursor across result rows. At a minimum, the cursor should support movement in a forward-only direction. Abstract base classes `DSISimpleResultSet`and `DSISimpleRowCountResult` are provided to deal with some basic functionality.

Related Topics

Simba SDK C++ API Reference

Simba SDK Java API Reference

"Building Blocks for a DSI Implementation" on page 21

"Lifecycle of DSI Objects" on page 96

API Overview

This section introduces the functionality and workflows of the C++ DSI API and the DSI API Extensions, which are the main APIs that you use to build a custom connector. The Java APIs are similar.

DSI API

The DSI API exposes the classes needed to build your own Data Store Interface Implementation using C++. The C# and Java versions of these classes, the DotNet DSI API and the Java DSI API, provide similar functionality as the C++ classes.

The DSI API functionality is grouped into Core classes and Data Engine classes.

Core classes

The Core classes provide all of the essential functionality to establish and manage the connection to your data source:

Class	Description
`IDriver`	`IDriver` is a singleton instance constructed when the connector is first loaded. Its primary responsibility is to construct `IEnvironment` objects and manage any connector-wide properties. An abstract base class `DSIDriver` is provided to assist in some of these responsibilities, including initializing defaults and managing properties.
`IEnvironment`	`IEnvironment` objects correspond to the ODBC environment (ENV) handles allocated by `SQLAllocHandle`. Their primary responsibility is to construct `IConnection` objects and manage any environment properties. An abstract base class `DSIEnvironment` is provided.
`IConnection`	`IConnection` objects correspond to the ODBC connection (DBC) handles allocated by `SQLAllocHandle`. Their primary responsibility is to handle user authentication, construct `IStatement`objects, and manage any connection properties. An abstract base class `DSIConnection` is provided.
`IStatement`	`IStatement` objects correspond to the ODBC statement (STMT) handles allocated by `SQLAllocHandle`. Their primary responsibility is to construct `IDataEngine` objects and manage any statement properties. An abstract base class `DSIStatement` is provided.
`IMessageSource`	`IMessageSource` is responsible for loading error messages and warnings from your connector. An abstract implementation `DSIMessageSource` is provided to load messages generated by the SDK. For more information, see "*Using or building a message source*" in "Handling Errors and Exceptions" on page 77.

Class	Description
`ILogger`	`ILogger` is responsible for storing or printing log messages from your connector. Each of the `IDriver`, `IEnvironment`, `IConnection`, and `IStatement` classes has a `GetLog()` method which must return the most appropriate logger for that object. You may share loggers between all the objects or construct a different logger for each. The `DSIFileLogger` class is fully implemented to store the log messages to a text file, but you may change the behaviour in any way by extending the `ILogger` interface directly or by subclassing the partially implemented `DSILogger`class.

Data Engine classes

The Data Engine classes are the subset used to perform the data access functions against your data store:

Class	Description
`IDataEngine`	`IDataEngine` is responsible for constructing an `IQueryExecutor` when preparing queries or constructing an `IResult` for catalog function metadata. An abstract base class `DSIDataEngine` is provided to assist in implementing filters for the catalog function metadata.
`IQueryExecutor`	`IQueryExecutor` is responsible for executing a query and generating `IResults` objects.
`IResults`	An `IResults` object represents a collection of one or more `IResult` objects. `DSIResults` provides a basic implementation for accessing and managing a collection of `IResult` objects.
`IResult`	`IResult` is responsible for retrieving column data and maintaining a cursor across result rows. At a minimum, the cursor should support movement in a forward-only direction. Abstract base classes `DSISimpleResultSet`and `DSISimpleRowCountResult` are provided to deal with some basic functionality.

Related Topics

Simba SDK C++ API Reference

Simba SDK Java API Reference

"Building Blocks for a DSI Implementation" on page 21

"Lifecycle of DSI Objects" on the next page

Lifecycle of DSI Objects

The objects of the DSI API have a lifecycle that is modeled on, though not exactly the same as, the lifecycle of ODBC handles. This section explains the lifecycle in the C++ SDK for ODBC connectors.

The `IDriver` object is instantiated when the connector is loaded, and a single instance is alive until the connector is unloaded.

The `IDriver` object creates an `IEnvironment` when an application allocates environment handles. `IDriver` can create multiple `IEnvironment` objects. These are guaranteed to have been destroyed by the time the `IDriver` is destroyed.

`IEnvironment` create `IConnections`, which are guaranteed to have been destroyed by the time the parent `IEnvironment` has been destroyed. `IConnections` can be created and freed when an application chooses, but are typically long-lived objects, with multiple actions occurring before being destroyed.

`IConnections` create `IStatements`, which are guaranteed to have been destroyed by the time the parent `IConnection` has been destroyed. `IStatements` can be short- or long-lived objects depending on the application. If the application re-uses statements, then they tend to be long-lived, while if the application does not re-use statements they tend to be short-lived.

`IStatements` create `IDataEngines`, which are guaranteed to have been destroyed by the time the parent `IStatement` has been destroyed.

`IDataEngines` create `IQueryExecutors`, which are guaranteed to have been destroyed by the time the parent `IDataEngine` has been destroyed. `IQueryExecutors` have a lifespan that matches the lifespan of a prepared and executed, or directly executed, query. A single `IQueryExecutor` is used for multiple executions of a prepared query.

Any objects created by an `IQueryExecutor` are guaranteed to have been destroyed by the time the parent `IQueryExecutor` has been destroyed.

`IQueryExecutors` create `IResults`, which are destroyed by the `IQueryExecutors` that created them. As stated above, `IResults` are guaranteed to have been destroyed before the `IQueryExecutor`.

`IResult` objects are accessed through `IResults` objects. However, the timing of their creation and destruction is determined by a connector's implementation. The `DSIResults` implementation creates `IResult` objects during construction and destroys them during destruction. Note that an `IResult` object is not accessible after it has been destroyed by the parent `IResults`object.

Related Topics

"API Overview" on page 94

Working With the Java API

This section describes the features in the Simba SDK that are specific to the Java API.

JDBC Time and Timestamp with Timezone

JDBC exposes the time and timestamp types with timezone information, represented as a `Calendar` object. If your data store supports timezone information for these types, it can be accessed by the `TimeTz` and `TimestampTz` types, both for insertion and retrieval.

To supply timezone information when retrieving data, instead of using the normal `java.sql.Time` or `java.sql.Timestamp` types, use the supplied `com.simba.dataengine.utilities.TimeTz` or `com.simba.dataengine.utilities.TimestampTz` types. These types are essentially a pair of the datetime class, along with a `Calendar` that supplies the timezone information. The SDK will automatically perform the correct operations to interpret that data when it passes it to applications.

To use timezone information when inserting data, use the `getTimeTz()` and `getTimestampTz()` methods of the `DataWrapper` class to get the classes which hold both the datetime types and the `Calendar` holding the timezone information. If your data store does not support timezones for the datetime types, calling the normal `getTime()` and `getTimestamp()` methods will automatically convert the datetime types to the local timezone.

JDBC Updatable ResultSets

JDBC provides the functionality to modify result sets that are generated from statements. SimbaJDBC allows you to add this functionality to your JDBC connector, if your data source supports it, by making the following changes to your CustomerDSII:

1. Set the DSI_SUPPORTS_UPDATABLE_RESULT_SETS property in your `CustomerDSIIConnection` object to a combination of DSI_SUPPORTS_URS_INSERT, DSI_SUPPORTS_URS_DELETE, and DSI_SUPPORTS_URS_UPDATE, depending on the extent of the modifications you will support on your result set.
2. Override and implement the following virtual methods:

a. `appendRow()` - Add a new empty row to the end of the result set.

b. `deleteRow()` - Delete the row at the current cursor position.

c. `writeData()` - Write data to the specified cell in the current row.

3. The following virtual methods from `IResultSet` should also be overridden and implemented. However, you may choose to return `false` if this information is not available:

a. `rowDeleted()` - Determine if the current row has been deleted.

b. `rowInserted()` - Determine if the current row has been inserted.

c. `rowUpdated()` - Determine if the current row has been updated.

4. The following virtual methods from `IResultSet` may also optionally be overridden and implemented:

a. `onStartRowUpdate()` - Called before writing data to update a row. This is not called after `appendRow` because it is implied that data will be written.

b. `onFinishRowUpdate()` - Called after writing all updated or inserted data in a row.

Developing for different Versions of JDBC

Simba SDK includes implementations for building connectors that work with JDBC 4.0, 4.1, and 4.2. You can develop connectors for any of these versions of JDBC, or you can develop a 'hybrid' connector that works with multiple versions, instantiating the appropriate classes at runtime.

Interface Versions

This section lists the classes that have different versions in order to support the different versions of JDBC:

AbstractDataSource

- Use `JDBC4AbstractDataSource` for JDBC 4.0
- Use `JDBC41AbstractDataSource` for JDBC 4.1
- Use `JDBC42AbstractDataSource` for JDBC 4.2
- Use `HybridAbstractDataSource` for hybrid versions

AbstractDriver

- Use `JDBC4AbstractDriver` for JDBC 4.0
- Use `JDBC41AbstractDriver` for JDBC 4.1
- Use `JDBC42AbstractDriver` for JDBC 4.2
- Use `HybridAbstractDriver` for hybrid versions

ObjectFactory

- Use `JDBC4ObjectFactory` for JDBC 4.0
- Use `JDBC41ObjectFactory` for JDBC 4.1
- Use `JDBC42ObjectFactory` for JDBC 4.2
- Use `HybridJDBCObjectFactory` for hybrid versions

If you are upgrading the code for an existing connector developed using Simba SDK 9.1 or earlier, then you must rename and modify your implementations of `JDBCAbstractDataSource`, `JDBCAbstractDriver`, and `JDBCObjectFactory` to implement the appropriate classes listed in the table above. If you are upgrading from Simba SDK 9.4, you may need to remove support for JDBC 3.0 if your implementation has support for it. If you are creating a new connector, then determine the appropriate classes to implement from the table above.

Internally, the Simba SDK includes JDBC version-specific implementations for the various JDBC classes such as `SConnection`, `SDatabaseMetadata`, etc. Examples of these include `S3Connection`, `S4Connection`, etc. Each version of the `AbstractFactory` will therefore return the appropriate subclasses for its target JDBC version (e.g. `JDBC4ObjectFactory`'s `creationConnection()` method will return an `S4Connection` object. In the case of a hybrid connector, its factory will determine which classes to create at runtime as described in the next section.

Determing and Recording the JDBC Version at Runtime

When developing a hybrid connector, the connector must determine which version of JDBC is running, and pass this information to the Simba SDK via the `HybridAbstractDataSource` and `HybridAbstractDriver` classes.

While there are a number of techniques for determining the JDBC version at runtime, JavaUltralight checks the value of the 'MODE' parameter in the connection string. For information on the Java Ultralight sample connector, see "JavaUltralight Sample Connector" on page 20. The following example shows a connection string that includes this MODE parameter:

jdbc:simba://User=odbc_user;Password=odbc_user_password;MODE=JDBC4

If the MODE parameter does not exist, JavaUltraLight detects its absence and then assumes that JDBC 4.0 should be used.

JavaUltralight provides an example of determining the version using the connection string. Its `HybridUtilities` class contains a static method that looks for this parameter and, if found, returns the appropriate JDBC version enum:

```
public final class HybridUtilities
{
  public static JDBCVersion runningJDBCVersion(String modeProperty)
  {
    if ((null != modeProperty) && (modeProperty.equals("JDBC42")))
    {
      return JDBCVersion.JDBC42;
    }
    else if ((null != modeProperty) &&
(modeProperty.equals("JDBC41")))
    {
      return JDBCVersion.JDBC41;
    }
    else
    {
    return JDBCVersion.JDBC4;
    }
  }
}
```

Once the JDBCVersion enum version is determined, it must then be passed to the Simba SDK by implementing the `runningJDBCVersion()` methods when subclassesing `HybridAbstractDriver` and `HybridDataSource`.

The following example shows how JavaUltralight's `ULJDBCHybridDriver` and `ULJDBCHybridDataSource` classes use the static `HybridUtilities::runningJDBCVersion()` method, described above, to pass this information to the Simba SDK:

```
public class ULJDBCHybridDriver extends HybridAbstractDriver
{
```

```
        private String m_mode = null;
        protected Pair<IConnection, ConnSettingRequestMap> getConnection( Properties info)
        throws SQLException
        {
                Pair<IConnection, ConnSettingRequestMap> result = super.getConnection
                (info);
                ConnSettingRequestMap connectionProperties = result.value();
                if ((null != connectionProperties) && (null !=
                connectionProperties.getProperty(ULPropertyKey.MODE)))
                {
                        m_mode = connectionProperties.getProperty(
                ULPropertyKey.MODE).getString();
                }
                return result;
        }
        protected JDBCVersion runningJDBCVersion()
                {
                        return HybridUtilities.runningJDBCVersion(m_mode);
                }
        }
        public class ULJDBCHybridDataSource extends HybridAbstractDataSource
        {
                protected JDBCVersion runningJDBCVersion()
        {
                return HybridUtilities.runningJDBCVersion(getCustomProperty(
                ULPropertyKey.MODE));
        }
}
```

Connector Auto-Loading

To allow for the auto-loading of a JDBC 4.0, JDBC 4.1, JDBC 4.2, or hybrid connector, you must have the file `METAINF/services/java.sql.Driver` containing the connector class to load in this .jar.

To create the file using Ant, add the following `Service` tag to the .jar tag in the connector's `.xml`build file:

```
<service type="java.sql.Driver" provider="your.driver.class.name"/>
```

For example, JavaUltralight's `JavaUltraLightBuilder.xml` build file specifies the following for JDBC 4.0 and hybrid builds respectively:

```
<service type="java.sql.Driver" provider="com.simba.ultralight.core.jdbc4.ULJDBC4Driver"/>
<target name="JavaUltraLightBuildDebug4"
  .
  .
  <jar jarfile="${jardest}/${JavaUltraLight4Jar}" basedir="${dest}" includes="com/simba/**">
     <service type="java.sql.Driver"
          provider="com.simba.ultralight.core.jdbc4.ULJDBC4Driver"/>
  </jar>
</target>
<target name="JavaUltraLightBuildDebugHybrid"
      depends="JavaUltraLightCompileDebugHybrid, UnjarHybrid"
      description="generate the Java UltraLight Jar file in debug mode">
      <mkdir dir="${jardest}"/>
  .
  .
  <jar jarfile="${jardest}/${JavaUltraLightHybridJar}"
   basedir="${dest}"
   includes="com/simba/**">
     <service type="java.sql.Driver" provider="com.simba.ultralight.core.hybrid.ULJDBC4Driver"/>
  </jar>
</target>
```

To auto-load the connector in your application, simply pass in `jdbc:simba://localhost` along with the user name and password as the URL, as shown in the following code example:

```
String url = "jdbc:simba://localhost;UID=username;PWD=test;";
m_connection = DriverManager.getConnection(url);
```

JDBC 4.0, 4.1, and 4.2 Exceptions

Exceptions created by the connector generate a `SQLException` by default. To generate an exception specific to JDBC 4.0, 4.1, or 4.2, specify the exception type when calling `createGeneralException()` as shown in the following example:

ULDriver.s_ULMessages.createGeneralException(DSIMessageKey.NOT_IMPLEMENTED.name(), ExceptionType.INTEGRITY_CONSTRAINT_VIOLATION);

Pooled Connections

A pooled connection can be created by calling `JDBCObjectFactory::createPooledConnection()`. If any specific behaviour is required, a connector can optionally override `createPooledConnection()` to return a subclass of `PooledConnection`. The three classes provided by Simba SDK for the respective JDBC versions each return the appropriate version of 'SPooledConnection' by default. For example, `JDBC4ObjectFactory::createPooledConnection()` returns an `S4PooledConnection` as shown in the following example:

```
/**
    * Attempts to establish a physical database connection that can be used as a pooled connection.
    * @param connection The connection to use to create the
    * <code>PooledConnection</code>.
    * @return A <code>PooledConnection</code> object that is a physical connection to the database that this <code>ConnectionPoolDataSource</code> object represents.
    * @throws SQLException    If a database access error occurs.
*/
protected PooledConnection createPooledConnection(SConnection connection) throws SQLException
{
   return new S4PooledConnection(connection);
}
```

Setting and Initializing Client Information

If a connector uses non-standard client info properties, both the initialization (e.g. loading) and the setting of these properties must be handled by the connector's connection class. These tasks are handled in the `loadClientInfoProperties()` and`setClientInfoProperty()` methods of the connection as shown in the following example from JavaUltraLight:

```
private void loadClientInfoProperties() throws ErrorException
{
        // TODO #XX: Define your custom client info properties.
        // Standard client info properties are Application_name, Client_user and
        // client_hostname.
        // Other client info properties have to be defined here
        ClientInfoData fakeCustomCLientInfo = new ClientInfoData(
```

```
        ULClientInfoPropertyKey.UL_CUSTOM_CLIENT_INFO,
        25,
        "FakeCustomClientInfoForUltralight",
        "Just a fake client info property to show how to define them.");
        setClientInfoProperty(fakeCustomCLientInfo);
}
public void setClientInfoProperty(String propName, String propValue)throws ClientInfoException
{
        // Check that the property name is valid and store the new property
        // values.
        super.setClientInfoProperty(propName, propValue);
        // TODO: Implements the wanted behaviour
        // Usually the connector stores the value specified in a suitable location
        // in the database.
        // For example in a special register, session parameter, or system table
        // column.
        LogUtilities.logInfo(
        String.format("Property {0} has now the value {1}", propName,
        propValue), m_log);
}
```

Handling Deregistration

JDBC 4.2 introduced the new `DriverAction` interface allowing JDBC connectors to be notified when they are being deregistered by the JDBC DriverManager. Implementing this interface allows connectors to handle the notification and perform clean up tasks such releasing resources. Note that the implementation should not perform the deregistration, but rather, perform any clean up required while the connector is being deregistered by the DriverManager.

Simba SDK 10.3 exposes this notification via the `IDriver` interface and a default implementation is provided in the `DSIDriver` class which does nothing. If you need to handle the deregisration event to perform clean up tasks, implement the `deregister()` method in your `DSIDriver`-derived class.

Related Topics

"API Overview" on page 94

"Lifecycle of DSI Objects" on page 96

"Sample Connectors and Projects" on page 18

"JavaUltralight Sample Connector" on page 20

Data Types

The Simba SDK provides a data type to handle each of the types in the SQL specification. This section lists the types for each SDK, and includes instructions on how to convert the types from your data store into the Simba SDK data types.

In the C++ SDK, you can also create your own custom C and SQL data types.

SQL Data Types in the C++ SDK

`SqlData` objects represent the SQL types and encapsulate the data in a buffer. When you have a `SqlData` object and would like to know what data type it is representing, you can use `GetMetadata()->GetSqlType()` to retrieve the associated `SQL_[TYPE]` type. For more information, see the file `SqlData.h`.

Fixed Length Types

The structures used to store the fixed-length data types represented by `SqlData` objects are listed below:

SQL Type	Simba SDK Data Type
SQL_BIT	simba_uint8
SQL_BIGINT (signed)	simba_int64
SQL_BIGINT (unsigned)	simba_uint64
SQL_DATE	TDWDate
SQL_DECIMAL	TDWExactNumericType
SQL_DOUBLE	simba_double64
SQL_FLOAT	simba_double64
SQL_GUID	TDWGuid
SQL_INTEGER (signed)	simba_int32
SQL_INTEGER (unsigned)	simba_uint32

SQL Type	Simba SDK Data Type
SQL_INTERVAL_DAY	TDWSingleFieldInterval
SQL_INTERVAL_DAY_TO_HOUR	TDWDayHourInterval
SQL_INTERVAL_DAY_TO_MINUTE	TDWDayMinuteInterval
SQL_INTERVAL_DAY_TO_SECOND	TDWDaySecondInterval
SQL_INTERVAL_HOUR	TDWSingleFieldInterval
SQL_INTERVAL_HOUR_TO_MINUTE	TDWHourMinuteInterval
SQL_INTERVAL_HOUR_TO_SECOND	TDWHourSecondInterval
SQL_INTERVAL_MINUTE	TDWSingleFieldInterval
SQL_INTERVAL_MINUTE_SECOND	TDWMinuteSecondInterval

SQL Type	Simba SDK Data Type
SQL_INTERVAL_MONTH	TDWSingleFieldInterval
SQL_INTERVAL_SECOND	TDWSecondInterval
SQL_INTERVAL_YEAR	TDWSingleFieldInterval
SQL_INTERVAL_YEAR_TO_MONTH	TDWYearMonthInterval
SQL_NUMERIC	TDWExactNumericType
SQL_REAL	simba_double32
SQL_SMALLINT (signed)	simba_int16
SQL_SMALLINT (unsigned)	simba_uint16
SQL_TIME	TDWTime
SQL_TIMESTAMP	TDWTimestamp
SQL_TINYINT (signed)	simba_int8
SQL_TINYINT (unsigned)	simba_uint8
SQL_TYPE_DATE	TDWDate
SQL_TYPE_TIME	TDWTime

SQL Type	Simba SDK Data Type
SQL_TYPE_TIMESTAMP	TDWTimestamp
SQL_TYPE_DATE	TDWDate
SQL_TYPE_TIME	TDWTime
SQL_TYPE_TIMESTAMP	TDWTimestamp

Date, Time and DateTime Types

The associated SQL types for date, time, and datetime are listed below:

Type	SQL Type for ODBC 3.x
date	SQL_TYPE_DATE
time	SQL_TYPE_TIME
datetime	SQL_TYPE_TIMESTAMP

> **Important:**
> **Important:**
>
> SQL_DATE, SQL_TIME and SQL_TIMESTAMP are ODBC 2.x types, while SQL_TYPE_DATE, SQL_TYPE_TIME, and SQL_TYPE_TIMESTAMP are ODBC 3.x types. Since you are developing an ODBC 3.x connector, use the ODBC 3.x types.

Example: Simple Fixed-Length Data

The `SQLData` for a SQL_INTEGER contains a `simba_int32` type. This example shows you how to copy your integer value into the `simba_int32` type.

```
switch (in_data->GetMetadata()->GetSqlType())
{
  case SQL_INTEGER:
  {
    simba_int32 value = 1234;
```

```
    *reinterpret_cast<simba_int32*>(in_data->GetBuffer()) = value;
  }
}
```

Variable Length Types

The following variable-length data types are stored in buffers and represented by `SqlData` objects:

SQL Type	Data Type
SQL_BINARY	simba_byte*
SQL_CHAR	simba_char*
SQL_LONGVARBINARY	simba_byte*
SQL_LONGVARCHAR	simba_char*
SQL_VARBINARY	simba_byte*
SQL_VARCHAR	simba_char*
SQL_WCHAR	simba_byte*
SQL_WLONGVARCHAR	simba_byte*
SQL_WVARCHAR	simba_byte*

Note:

You can use `DSITypeUtilities::OutputWVarCharStringData` and `OutputVarCharStringData` for setting character data.

Example: Variable-Length Data

In the example below, the SQL_CHAR case shows how to use the type utilities, while the SQL_VARCHAR case shows how to use `memcpy`.

Note:

- In your custom connector code, SQL_CHAR, SQL_VARCHAR and SQL_LONGVARCHAR do not require separate cases.
- You custom connector code has other considerations, such as handling offsets in the data.

```
switch (in_data->GetMetadata()->GetSqlType())
{
  case SQL_CHAR:
```

```
  {
    simba_string stdString(“Hello”);
    return DSITypeUtilities::OutputVarCharStringData(
      &stdString,
      in_data,
      in_offset,
      in_maxSize);
  }
  case SQL_VARCHAR:
  {
    simba_string stdString("Hello");
    simba_uint32 size = stdString.size();
  in_data->SetLength(size);
  memcpy(in_data->GetBuffer(), stdString, size);
return false;
  }
}
```

SQL DataTypes in the Java SDK

This section explains the mapping between SQL types and the Simba SDK data types for JDBC.

Note:

Because Java does not support unsigned types, SQL types that have both unsigned and signed variations are mapped to the next largest data type.

SQL Type	Data Type
SQL_BIGINT (signed)	java.math.BigInteger
SQL_BIGINT (unsigned)	java.math.BigInteger
SQL_BINARY	byte[]
SQL_BIT	java.lang.Boolean
SQL_CHAR	java.lang.String

SQL Type	Data Type
SQL_DECIMAL	java.math.BigDecimal
SQL_DOUBLE	java.lang.Double
SQL_FLOAT	java.lang.Double
SQL_INTEGER (signed)	java.lang.Long
SQL_INTEGER (unsigned)	java.lang.Long
SQL_INTERVAL_DAY	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_DAY_TO_HOUR	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_DAY_TO_MINUTE	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_DAY_TO_SECOND	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_HOUR	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_HOUR_TO_MINUTE	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_HOUR_TO_SECOND	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_MINUTE	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_MINUTE_SECOND	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_MONTH	com.simba.dsi.dataengine.utilities.DSIMonthSpan
SQL_INTERVAL_SECOND	com.simba.dsi.dataengine.utilities.DSITimeSpan
SQL_INTERVAL_YEAR	com.simba.dsi.dataengine.utilities.DSIMonthSpan
SQL_INTERVAL_YEAR_TO_MONTH	com.simba.dsi.dataengine.utilities.DSIMonthSpan
SQL_LONGVARBINARY	byte[]
SQL_LONGVARCHAR	java.lang.String
SQL_NUMERIC	java.math.BigDecimal
SQL_REAL	java.lang.Float

SQL Type	Data Type
SQL_SMALLINT (signed)	java.lang.Integer
SQL_SMALLINT (unsigned)	java.lang.Integer
SQL_TINYINT (signed)	java.lang.Short
SQL_TINYINT (unsigned)	java.lang.Short
SQL_TYPE_DATE	java.sql.Date
SQL_TYPE_TIME	java.sql.Time or com.simba.dsi.dataengine.utilities.TimeTz
SQL_TYPE_TIMESTAMP	java.sql.Timestamp or com.simba.dsi.dataengine.utilities.TimestampTz
SQL_VARBINARY	byte[]
SQL_VARCHAR	java.lang.String
SQL_WCHAR	java.lang.String
SQL_WLONGVARCHAR	java.lang.String
SQL_WVARCHAR	java.lang.String

Interval Conversions

Type Name	SQL Type	Parameters
INTERVAL DAY	SQL_INTERVAL_DAY	Whole Day Precision For example: INTERVAL DAY (3)
INTERVAL DAY TO HOUR	SQL_INTERVAL_DAY_TO_HOUR	Day Precision For example: INTERVAL DAY (2)
INTERVAL DAY TO MINUTE	SQL_INTERVAL_DAY_TO_MINUTE	Day Precision For example: INTERVAL DAY (2) TO MINUTE

Type Name	SQL Type	Parameters
INTERVAL DAY TO SECOND	SQL_INTERVAL_DAY_TO_SECOND	Day Precision, Fractional Seconds Precision For example: INTERVAL DAY (2) TO SECOND (3)
INTERVAL HOUR	SQL_INTERVAL_HOUR	Hour Precision For example: INTERVAL HOUR (3)
INTERVAL HOUR TO MINUTE	SQL_INTERVAL_HOUR_TO_MINUTE	Hour Precision For example: INTERVAL HOUR (2)
INTERVAL DAY TO SECOND	SQL_INTERVAL_HOUR_TO_SECOND	Hour Precision, Fractional Seconds Precision For example: INTERVAL HOUR (3) TO SECOND (4)
INTERVAL MINUTE	SQL_INTERVAL_MINUTE	Minute Precision For example: INTERVAL MINUTE(2)
INTERVAL MINUTE SECOND	SQL_INTERVAL_MINUTE_TO_SECOND	Minute Precision, Fractional Seconds Precision For example: INTERVAL MINUTE (3) SECOND (4)
INTERVAL MONTH	SQL_INTERVAL_MONTH	Month Precision For example: INTERVAL MINUTE(2)
INTERVAL SECOND	SQL_INTERVAL_SECOND	Whole Seconds Precision, Fractional Seconds Precision For example: INTERVAL SECOND (4,5)
INTERVAL YEAR	SQL_INTERVAL_YEAR	Year Precision For example: INTERVAL YEAR(3)
INTERVAL YEAR TO MONTH	SQL_INTERVAL_YEAR_TO_MONTH	Year Precision For example: INTERVAL YEAR(2) TO MONTH)

Adding Custom SQLDataType

Using the C++ Simba SDK, you can add custom SQLDataTypes to your DSII. Each custom data type that you add must be based on an existing data type. This allows applications to handle your custom types transparently without requiring additional logic.

The SQLite Sample driver demonstrates this to implement a Tweet custom data type as a fixed length character field combined with a length field.

This functionality is available to connectors that use the SQL Engine, and to those that do not.

Example:

You can add a type called `Money` that is based on `Numeric`, but is restricted to two decimal places. It may also contain a custom conversion to character types that adds the currency character.

Simba SDK uses the following classes to handle custom SQLDataTypes:

- `UtilityFactory` class create a `SqlTypeMetadataFactory` object, which creates the metadata about the custom types.
- `SqlDataFactory` creates the object which represents the custom type.
- `SqlConverterFactory` converts the custom type to other data types.

This functionality is explained more in the instructions below.

To Add a Custom SQLDataType:

Note:

Corresponding class and function names from the SQLite sample driver are noted in square brackets.

1. Modify your custom `DSIIDriver [SLDriver]` object to override and implement the virtual method `CreateUtilityFactory`. In this method, return a `CustomerDSIIUtilityFactoryClass[SLUtilityFactory]`. This class provides the other factories that implement custom data type behavior.
2. Create a `CustomerDSIIUtilityFactory [SLUtilityFactory]` class, which subclasses `Simba::Support::UtilityFactory`. This factory class provides classes that handle the custom type metadata, data, and conversion of the custom data types.
   a. `CreateSqlConverterFactory()` creates a factory to create converters that convert custom data types to other types.
   b. `CreateSqlDataFactory()` creates a factory to create the actual SqlData objects that represent the custom data types.
   c. `CreateSqlTypeMetadataFactory()` creates a factory to create the metadata about the custom data types.
3. Create a `CustomerDSIISqlConverterFactory [SLSqlConverterFactory]` class which subclasses `Simba::Support::SqlConverterFactory`, and override and implement the following virtual methods:
   a. `CreateNewCustomSqlToCConverter()` - Takes a `SqlData` and `SqlCData` object representing the source and target types, and an `IWarningListener` for posting any conversion warnings to. The returned converter `[SLCustomTypeTweetConverter]` is responsible for converting from the source SQL data type to the target C data type.

   b. `CreateNewCustomCToSqlConverter()` - Takes a `SqlCData` and `SqlData` object representing the source and target types, and an `IWarningListener` for posting any conversion warnings to. The returned converter `[SLCustomTypeTweetConverter]` is responsible for converting from the source C data type to the target SQL data type.

4. Create a `CustomerDSIISqlDataFactory [SLSqlDataFactory]` class which subclasses `Simba::Support::SqlDataFactory`, and override and implement the following virtual methods:
   a. `CreateNewCustomSqlData()` - Takes a `SqlTypeMetadata` object representing a SQL data type, which is used to determine what SqlData object to create. Return a subclass of SqlData that represents the custom type `[SLTweetSqlData]` if supported, otherwise return NULL.

5. Create a `CustomerDSIISqlTypeMetadataFactory [SLSqlTypeMetadataFactory]` class which subclasses `Simba::Support::SqlTypeMetadataFactory`, and override and implement the following virtual methods:
   a. `CreateNewCustomSqlTypeMetadata()` - Create a new `SqlTypeMetadata` object that represents the custom data type specified. The helper function `SetupStandardMetadata` is provided to set up the standard type metadata for the standard SQLDataTypes. Return NULL if the specified type is not supported.
   b. `SetCustomTypeDefaults()` - Set the default metadata for the specified data type on the specified `SqlTypeMetadata` object. This allows for reuse of existing `SqlTypeMetadata` objects, rather than creating new objects.

6. Ensure that the custom data types are reported in the metadata source for type information. In particular, the DSI_USER_DATA_TYPE_COLUMN_TAG should return the custom type identifier.

ODBC Custom C Data Types

Using the C++ Simba SDK, you can add custom C data types to your DSII. Each custom data type that you add must be based on an existing data type. This allows applications to handle your custom types transparently without requiring additional logic.

This functionality is available to connectors that use the SQL Engine, and to those that do not.

Simba SDK uses a UtilityFactory class to create a `SqlCTypeMetadataFactory` object to create the metadata about the custom types, then use and a `SqlConverterFactory` to convert the custom type to other data types.

The SQLite Sample driver demonstrates this by implementing a Tweet custom data type as a fixed length character field combined with a length field.

To Add Custom C Data Types:

Note:

Corresponding class and function names from the SQLite sample driver are noted in square brackets.

1. Create a header file to package with your ODBC connector. In this header file, define the type ID for your custom C type, field ID's for any custom metadata fields, and the struct of your custom C data type. Note that field ID's must start at 0x4100.

2. Modify your `CustomerDSIIDriver [SLDriver]` object to override and implement the virtual method `CreateUtilityFactory()` to return a `CustomerDSIIUtilityFactoryClass`

`[SLUtilityFactory]`. This class will provide the other factories that implement custom data type behavior.

3. Create a `CustomerDSIIUtilityFactory [SLUtilityFactory]` class that subclasses `Simba::Support::UtilityFactory`. This factory class will provide classes that handle the custom type metadata, data, and conversion of the custom data types.
   a. `CreateSqlConverterFactory()` creates a factory to create converters that convert custom data types to other types.
   b. `CreateSqlCDataTypeUtilities()` creates a utility class which describes the custom C data types.
   c. `CreateSqlCTypeMetadataFactory()` creates a factory to create the metadata about the custom data types.

4. Create a `CustomerDSIISqlConverterFactory [SLSqlConverterFactory]` class which subclasses `Simba::Support::SqlConverterFactory`, and override and implement the following virtual methods:
   a. `CanConvertCustomCTypeToSql()` takes the ID of a custom C data type and the TDWType enum of the target SQL type to convert to, and determines if the type conversion can be performed.
   b. `CanConvertSqlToCustomCType()` takes the TDWType enum of a SQL data type and the ID of a custom C data type to convert to, and determines if the type conversion can be performed.
   c. `CreateNewCustomSqlToCConverter()` takes a SqlData and SqlCData object representing the source and target types, and an IWarningListener for posting any conversion warnings to. The returned converter `[SLCustomTypeTweetConverter]` is responsible for converting from the source SQL data type to the target C data type.
   d. `CreateNewCustomCToSqlConverter()` takes a SqlCData and SqlData object representing the source and target types, and an IWarningListener for posting any conversion warnings to. The returned converter `[SLCustomTypeTweetConverter]` is responsible for converting from the source C data type to the target SQL data type.

5. Create a `CustomerDSIISqlCDataTypeUtilities [SLCDataTypeUtilities]` class which subclasses `SqlCDataTypeUtilities`, and override and implement the following two methods:
   a. `IsSupportedCustomType()` takes in the ID of a type and determines if it is a valid custom C data type.
   b. `GetStringForCType()` takes is in the ID of a C data type and returns the string representation of it.

   Optionally override the following two methods if the custom C data type will support custom metadata fields:
   a. `IsSupportedCustomMetadataField()` takes in the ID of a field identifier, along with the field's indent and determines if the field identifier and indent are valid.
   b. `GetCustomMetadataFieldType()` takes in the field indent and returns the data type that it represents.

   Note:

   When you override a function, your custom function should defer to the implementation of the parent class when the ID of a non-custom type is passed in. That is, your custom function should only handle your custom types.

6. Create a `CustomerDSIISqlCTypeMetadataFactory [SLCTypeMetadataFactory]` class which subclasses `Simba::Support::SqlCTypeMetadataFactory`, and override and implement the following virtual methods:
   a. `CreateNewCustomSqlCTypeMetadata()` creates a new `SqlCTypeMetadata` object that represents the custom C data type specified.
   b. `ResetCustomTypeDefaults()` sets the default values for the custom C data type.

   Optionally create `CustomerSqlCTypeMetadata` which subclasses `SqlCTypeMetadata` if custom metadata fields are required for the custom C data type. This object will then be constructed and returned by the `CreateNewCustomSqlCTypeMetadata()` method.

   In `CustomerSqlCTypeMetadata`, the `SetField()/GetField()` methods must be overridden if there are custom metadata fields to set/get.

Specifications

This section lists the platform and compiler requirements for the Simba SDK. It also lists the level of SQL conformance that is supported.

Supported Platforms

This section lists the platforms and compilers that are supported by the Simba SDK version 10.3.

Hardware Requirements

On all supported platforms, the minimum hardware requirements are as follows:

- 8 GB of free disk space
- 1 GB RAM

Software Requirements

The following table lists the supported platforms and compilers:

Platform	Versions	Compilers	Bits
Windows	10 & 11 Server 2016, 2019 & 2022	Visual Studio 2019 & 2022 .NET Standard 2.0 .NET Core .NET Framework 3.5 & 4.6.2	32, 64
Linux	CentOS/Oracle Linux/RHEL 8 SLES 15 Ubuntu 24.04	GNU GCC 8.5	32, 64
Linux ARM	RHEL 8	GNU GCC 8.5	32, 64
macOS	11 (Apple M1) 13	Xcode 12.4	64 64 64
Solaris SPARC	10, 11	Oracle Solaris Studio 12.6 (Solaris 11)	32, 64

Platform	Versions	Compilers	Bits
Solaris x86	11	Oracle Solaris Studio 12.6 (Solaris 11)	32, 64
	7.2	XLClang C/C++ V16.1	32, 64

JDBC and JDK Support

The following list shows the JDK requirements for each version of JDBC:

- JDBC 4.2 used with JDK 1.8

Supported ODBC/SQL Functions

This section lists the ODBC-defined scalar functions that are supported by the SQL Engine.

Explicit Covert functions

- CONVERT
- CAST

String Functions

- ASCII
- CHAR
- CONCAT
- INSERT
- LCASE
- LEFT
- LENGTH
- LOCATE
- LTRIM
- REPEAT
- REPLACE
- RIGHT

- RTRIM
- SOUNDEX
- SPACE
- SUBSTRING
- UCASE

System Functions

- DATABASE
- IFNULL
- USER

Numeric Functions

- ABS
- ACOS
- ASIN
- ATAN
- ATAN2
- CEILING
- COS
- COT
- DEGREES
- EXP
- FLOOR
- LOG
- LOG10
- MOD
- PI

- POWER
- RADIANS
- RAND
- ROUND
- SIGN
- SIN
- SQRT
- TAN
- TRUNCATE

Aggregate Functions

- AVG
- COUNT
- MAX
- MIN
- STDDEV
- STDDEV_POP
- SUM
- VAR
- VAR_POP

Time, Date, and Interval Functions

- CURDATE
- CURTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIME (time precision)

- CURRENT_TIMESTAMP
- CURRENT_TIMESTAMP (time precision)
- DAYNAME
- DAYOFMONTH
- DAYOFWEEK
- DAYOFYEAR
- HOUR
- MINUTE
- MONTH
- MONTHNAME
- NOW
- QUARTER
- SECOND
- TIMESTAMPADD
- TIMESTAMPDIFF
- WEEK
- YEAR

Supported SQL Conformance Level

The Simba SDK supports the full core-level ODBC 3.80. It supports most of the Level 1 and Level 2 API.

The ODBC specification provides three levels of SQL grammar conformance: Minimum, Core and Extended. Each higher level provides more fully implemented data definition and data manipulation language support. The level of supported SQL grammar is dependent on your SQL-enabled data source. At the very least, your SQL-enabled data source must conform to the minimum SQL grammar defined by the ODBC version 3.52 specification.

Conformance Level	Interfaces	Conformance Level	Interfaces
Core	SQLAllocHandle	Core	SQLGetInfo

Conformance Level	Interfaces	Conformance Level	Interfaces
Core	SQLBindCol	Core	SQLGetStmtAttr
Core	SQLBindParameter	Core	SQLGetTypeInfo
Core	SQLCancel	Core	SQLNativeSql
Core	SQLCloseCursor	Core	SQLNumParams
Core	SQLColAttribute	Core	SQLNumResultCols
Core	SQLColumns	Core	SQLParamData
Core	SQLConnect	Core	SQLPrepare
Core	SQLCopyDesc	Core	SQLPutData
Core	SQLDescribeCol	Core	SQLRowCount
Core	SQLDisconnect	Core	SQLSetConnectAttr
Core	SQLDriverconnect	Core	SQLSetCursorName
Core	SQLEndTran	Core	SQLSetDescField
Core	SQLExecDirect	Core	SQLSetDescRec
Core	SQLExecute	Core	SQLSetEnvAttr
Core	SQLFetch	Core	SQLSetStmtAttr
Core	SQLFetchScroll	Core	SQLSpecialColumns
Core	SQLFreeHandle	Core	SQLStatistics
Core	SQLFreeStmt	Core	SQLTables
Core	SQLGetConnectAttr	Level 1	SQLBrowseConnect
Core	SQLGetCursorName	Level 1	SQLMoreResults
Core	SQLGetData	Level 1	SQLPrimaryKeys

Conformance Level	Interfaces	Conformance Level	Interfaces
Core	SQLGetDescField	Level 1	SQLProcedureColumns
Core	SQLGetDescRec	Level 1	SQLProcedures
Core	SQLGetDiagField	Level 2	SQLColumnPrivileges
Core	SQLGetDiagRec	Level 2	SQLDescribeParam
Core	SQLGetEnvAttr	Level 2	SQLForeignKeys
Core	SQLGetFunctions	Level 2	SQLTablePrivileges

Methods

The following section contains guidelines and considerations for implementing specific methods in your connectors.

IStatement::ExecuteBatch()

This method is used to execute a set of statements in a batch.

Note:

This method can only be used to execute a statement batch coming from a JDBC client. See the documentation for Java's `java.sql.Statement#executeBatch()` for more information.

The statements in in_statements are not already converted to the underlying datas source's native syntax. If the IDriver property `DSI_DRIVER_NATIVE_SQL_BEFORE_PREPARE` is set to `DSI_PROP_TRUE`, the default implementation of this method transforms the statements with `IConnection::ToNativeSql()`.

All statements in in_statements should return a single rowcount result (no result sets).

The `DSI_CONN_STOP_ON_ERROR` connection property should be respected.

By default, the implementation runs according to the following logic:

Note:

This is not functioning code.

```
BatchResult res = new BatchResult();
for (stmt : in_statement)
        {
        try
        {
                res.AddRowCount(Execute(stmt));
        }
        catch (...)
        {
                res.AddError(GetCurrentException());
                if (StopOnError)
                break;
        }
}
return res;
```

Where `Execute(stmt)` calls `IStatement::CreateDataEngine()`, uses it to execute the statement via the query executor returned by `IDataEngine::Prepare()`, and then destroys the query executor and data engine. If the executed statement returns multiple results, or a resultset, this is represented as an error in the returned IBatchResult object.

Statements

@param in_statements

The list of SQL statements to execute as part of the batch.

@return

An IBatchResult object describing the results of the execution (OWN)

virtual `Simba::DSI::IBatchResult* ExecuteBatch(const std::vector<simba_wstring>& in_statements);`

Exposes an iterator to the results of `IStatement::ExecuteBatch()`.

Initially, this object is positioned before the first result, and may only be iterated over once.

This object's results are sequential. The first result is for the first statement in the batch, the next result is for the second statement, and so forth. There is at most 1 result per statement.

The function produces results for a contiguous prefix of the statements, and those results are either a single rowcount or a set of errors. Depending on the structure of the statements, this prefix may be the entirety of the set. No gaps occur in the results and, if there are no errors, there are as many results in the object as there were statements in the batch. If the DSII stops on errors within a batch (`DSI_CONN_STOP_ON_ERROR` is set), then fewer objects than statements can occur. The JDBC specification states a given data source must be consistent in this behavior, either always stopping on error or never stopping. The SDK does not enforce this.

Result

Returns IBatchResult object. This object includes the following interface.

IBatchResult() {}

Constructor

`virtual ~IBatchResult() {}`

Destructor

enum ResultType

Describes the state of this object.

ROWCOUNT_RESULT

Indicates this object is currently positioned on a rowcount result.

ERROR_RESULT

Indicates this object is currently positioned on an error result.

NO_MORE_RESULTS

Indicates this object has no more results.

virtual ResultType MoveNext() = 0;

Moves to the next result exposed by this object (if there are any). Returns `NO_MORE_RESULTS` if there are no more results, otherwise returns `ROWCOUNT_RESULT or ERROR_RESULT` to indicate the type of the current result.

virtual bool GetCurrentRowCount(simba_uint64& out_rowCount) const = 0;

Gets the current rowcount result. If the rowcount was known, it is returned via the out parameter out_rowCount.

Note:

This may only be called if the last call to `MoveNext()` returned `ROWCOUNT_RESULT`.

virtual const std::vector<ErrorException>& GetCurrentErrors() const = 0;

Gets any errors that occurred for the current result.

Note:

Will return an empty vector unless MoveNext() returned ERROR_RESULT.

Compiling Your Connector

The 5 Day Guides at http://www.simba.com/resources/sdk/documentation/ provide step-by-step instructions on how to compile and build the debug version of each sample connector. This section provides more details on the compile and build process, and explains the available options.

For information on packaging your connector as a product for end customers, see "Packaging Your Connector" on page 151.

Upgrading Your Makefile to 10.1

In the 10.1 release, the Simba SDK introduces a new, simplified makefile system which is very different from the ones used in previous versions. This section explains how to customize and upgrade the sample makefiles in SDK 10.1 for your own custom ODBC connectors.

Updated Name and Location of Makefiles

This section explains the name and location of the new makefiles, including how to invoke them when building your custom ODBC connector.

How to invoke the makefile

We recommend that you invoke the makefile using `[DriverFolder]/Source/mk.sh`. This script invokes and passes along all the arguments to the makefile, `[DriverFolder]/Source/GNUmakefile`.

Note:

We do not recommend using `[DriverFolder]/Source/GNUmakefile` directly, because all object files will be generated directly under the source directory.

The following table summarizes the differences in the makefiles between the 10.1 and 10.0 release.

10.1	10.0
How to invoke the makefile	
From the `[DriverFolder]/Source` folder, type: `./mk.sh`	From the `[DriverFolder]/Makefile` folder, type: `make -f [DriverName].mak`
Main entry makefile	
[DriverFolder]/Source/GNUmakefile	[DriverFolder]/Makefile/ [DriverName].mak

10.1	10.0
Supporting makefiles for each connector	
The content of the supporting makefiles for each connector is merged into the entry makefile.	These makefiles are invoked by the entry makefile: [DriverFolder]/Source/Makefile [DriverFolder]/Source/Makefile_FLAGS.mak [DriverFolder]/Source/Makefile_SRCS.mak
Common makefiles shared by all connectors	
These makefiles are invoked by the entry makefile: [SIMBAENGINE_DIR]/Makefiles/kit.mk [SIMBAENGINE_DIR]/Makefiles/kit.sh ▪ `Platform.mak`, which was used to obtain platform-dependent information, has been replaced by the new `kit.sh` script file. ▪ `Settings_XXX.mak`, `Rule_XXX.mak` and `Master_Targets.mak` have been merged into the new `kit.mk` file.	These makefiles are invoked by the entry makefile: [SIMBAENGINE_DIR]/Makefiles/Platform.mak [SIMBAENGINE_DIR]/Makefiles/Settings_[PlatformName].mak [SIMBAENGINE_DIR]/Makefiles/Master_Targets.mak [SIMBAENGINE_DIR]/Makefiles/Rules_[PlatformOrCompilerName].mak

Customizing the Sample Makefiles

The main entry makefile is `[DriverFolder]/Source/GNUmakefile`. In most cases, this is the only file that you need to modify for your custom ODBC connector. This section includes the following steps:

"Step 1: Modify the name and location of the generated binary files" below

"Step 2: Add source files and specify where to find them " on page 130

"Step 3: Add Search Paths for .h files and other compiler/linker flags" on page 132

Step 1: Modify the name and location of the generated binary files

1. Modify `target.driver = libQuickstart${BITS}.${SO}` and `target.server = QuickstartServer${BITS}` to change the default file name for the connector and server. Note the following:

- `${BITS}` represents the bitness of the current product, typically 32 or 64 (for OSX, it could also be 3264).
- `${SO}` is the default platform-dependendent extension name for shared library. Typically this is `dylib` for OSX and `so` for other UNIX systems.

2. Optionally, update the location. By default, the final product is built under `[DriverFolder]/Bin/[PlatformName]/[ConfigurationMode][BitNess]`, for example `[DriverFolder]/Bin/Linux_x86_gcc/debug64`. If this location needs to be changed, modify the `DESTDIR.bin` variable.

Example:

SDK 10.1 [DriverFolder]/Source/GNUmakefile

Define the product names

target.driver = libMyCustomDSII${BITS}.${SO}

target.server = MyCustomDSII${BITS}

#...

Change default install location

DESTDIR.bin = $./../MyDirectory/${PLATFORM}/${MODE}${BITS}

Comparing 10.0 and 10.1 variables for this step:

This table shows how the 10.1 variables in this step map to the 10.0 variables. In 10.0, the variables are in `[DriverFolder]/Source/Makefile`.

Description	Name in 10.1	Name in 10.0
name of connector	`target.driver`	`TARGET_SO`
name of server	`target.server`	`TARGET_BIN`
location of generated binary	`DESTDIR.bin`	`TARGET_BIN(SO)_PATH`
config mode (release/debug)	`MODE`	No equivalence
bitness (32/64/3264)	`BITS`	`BITNESS`
suffix of shared libs (so / dylib)	`SO`	`SO_SUFFIX`

In 10.0, `[DriverFolder]/Source/Makefile` uses `TARGET_BIN` and `TARGET_SO` to define binary file names. As well, destination directories are specified by `TARGET_BIN_PATH` and `TARGET_SO_PATH`, as shown in the following example:

Example:

SDK 10.0 [DriverFolder]/Source/Makefile

PROJECT = MyCustomDSII

MAKEFILE_PATH = ../Makefiles

ifeq ($(BUILDSERVER),exe)

TARGET_BIN_PATH = ../Bin/$(PLATFORM)

TARGET_BIN = $(TARGET_BIN_PATH)/$(PROJECT)_server_<TARGET>

else

TARGET_SO_PATH = ../Bin/$(PLATFORM)

TARGET_SO = $(TARGET_SO_PATH)/lib$(PROJECT)_<TARGET>.$(SO_SUFFIX)

endif

#...

Note:

In 10.0, the `_<TARGET>` suffix in `TARGET_BIN(SO)` is not a variable. Instead, this is a special string that was replaced by either a `_Debug` suffix or an empty string (depending on the config mode) in the master makefiles provided under SIMBAENGINE_DIR. By default, both debug and release connectors were generated into the same folder, and relied on suffixes in filenames to differentiate release and debug builds.

In 10.1, binary files do not have a `release` or `debug` suffix in their name. Instead, release and debug builds are put under different folders.

Step 2: Add source files and specify where to find them

1. Modify the file names. To do this, modify the following line to list your own source files:

${target}: Main_Unix.o QSConnection.o QSDataEngine.o...

Note:

This list actually contains object files, so they should all have `.o` extension, rather than their original `.cpp` extension names. As well, you do not need to include the paths to the source files.

2. Modify the target-specific files. If some source files should only be included when the DSII is built as a server, then you must add these files to a `${target.server}: ...` dependency list, instead of the common object file list `${target}: ...;` As well, when the DSII is build as a connector (a shared library), these files should be added to `${target.driver}: ...`

3. Modify the file paths. To do this modify the following line to include all directories that contain source files for the connector:

```
drvsrcdirs = $./Common  $./Core  $./DataEngine  $./DataEngine/Metadata
```

Note:

The symbol `$.` is a variable defined in `kit.mk` that represents the full path of the directory where GNUmakefile is located, for example `[DriverFolder]/Source/`. We recommend using this variable instead of using `[DriverFolder]/Source/`.

Example:

Suppose the source files of the `MyCustomDSII` connector are laid out in the following folder hierarchy:

Source

--/MySourceDir1

----MyCommonSource1.cpp # Common source files for both connector and server

----MyCommonSource2.cpp

----MyDriverSpecificSource1.cpp # Connector specific source file.

----MyCommonSource1.h # Common header files

----MyCommonSource2.h

----MyDriverSpecificSource1.h # Connector specific header file.

--/MyFolder2

----MyCommonSource3.cpp

----MyServerSpecificSource1.cpp # Server specific source file

----MyCommonSource3.h

----MyServerSpecificSource1.h # Server specific header file

--/MyIncludeDir1

----MyOtherInclude1.h

--/MyIncludeDir2

----MyOtherInclude2.h

In this case, the filename and path lists should be configured as follows:

```
makefile # SDK 10.1 [DriverFolder]/Source/GNUmakefile #... ### Specify
directories to all folders that contain source files for this connector
drvsrcdirs = $./MySourceDir1 $./MySourceDir2
```

4. Add the source file specific to "connector". For example:

${target.driver} : MyDriverSpecificSource1.o ${target.driver} : LDLIBS += $(call Mutual, ${CORESDK.a} ${SQLENGINE.a} ${ODBCSDK.a}) #...

5. Add source file specific to "server". For example:

```
${target.server} : MyServerSpecificSource1.o ${target.server} : CPPFLAGS +=
${SERVERSDK_CPPFLAGS} #...
```

6. Modify source file list shared by both "connector" and "server". For example:

${target} : MyCommonSource1.o MyCommonSource2.o MyCommonSource3.o #...

Comparing 10.0 and 10.1 variables:

The 10.1 variables involved in this step corresponds to variables in `[DriverFolder]/Source/Makefile_SRCS.mak` in 10.0:

Description	Name in 10.1	Name in 10.0
list of files	listed directly in target rules as .o	COMMON_SRCS
list of directories	drvsrcdirs	COMMON_SRCS

In 10.0, the list of source files, along with their paths, are specified by the `COMMON_SRCS` variable in `[DriverFolder]/Source/Makefile_SRCS.mak`.

Example:

SDK 10.0 [DriverFolder]/Source/Makefile_FLAGS.mak

Common Sources used to build this project.

COMMON_SRCS = \

Common/QSTableMetadataFile.cpp \

Common/TabbedUnicodeFileReader.cpp \

Core/QSConnection.cpp \

Core/QSDriver.cpp \

Core/QSEnvironment.cpp \

Core/QSStatement.cpp \

DataEngine/QSDataEngine.cpp \

DataEngine/QSMetadataHelper.cpp \

DataEngine/QSTable.cpp \

DataEngine/QSTypeInfoMetadataSource.cpp

Step 3: Add Search Paths for .h files and other compiler/linker flags

1. Add search paths for headers. To do this, modify the following line to include your own search directories for header files:

${target} : CPPFLAGS += $(addprefix -I, $. ${drvsrcdirs} $(patsubst %,%/Include, ${drvsrcdirs}) $./Setup)

In this sample makefile, all directories listed in `drvsrcdirs`, and an `Include` directory under each of those directories are automatically added as search directories. You may append additional directories if they are not already in this default list.

2. Add or modify other compiler and preprocessor flags other than the header file search paths. To do this, add or modify existing target dependency lists that contains `${target}: CXXFLAGS+=` and `${target}: CPPFLAGS+=...` respectively.

3. Modify the linker flags. To add or modify linker flags and thirdparty libraries to be linked, add or modify existing target dependency lists that contains `${target}: LDFLAGS+=` and `${target}: LDLIBS+=...`, respectively.

4. Modify the target specific flags. If a flag should be added when building connector but not server (or the other way around), then it should be listed under `${target.driver}` or `${target.server}`, instead of the common `${target}`. For example, `${target.server}: LDFLAGS+=...` means this LDFLAGS list only applies when building a server.

5. Modify the config mode specific flags. If a flag should be added only for either "release" or "debug", but not both, then users can append a `.release` or `.debug` suffix to the corresponding `XXXFLAGS` variable to allow such config mode specific flags. For example, `CXXFLAGS.release += -myflag1 indicates -myflag1` will only be added to `CXXFLAGS` in release mode. Similarly, `CPPFLAGS.debug += -DMY_MACRO and LDFLAGS.debug += -LMySearchPath` indicates `-DMY_MACRO` and `-LMySearchPath` will only be added to `CPPFLAGS` and `LDFLAGS` in debug mode.

For more information on implicit variables CXXFLAGS, CPPFLAGS, LDFLAGS, LDLIBS used in GNU make, see the GNU make documentation at https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html.

Example:

SDK 10.1 [DriverFolder]/Source/GNUmakefile`

Add a flag only for connector

#...

${target.driver} : CPPFLAGS += -DMY_DRIVER_MACRO

Add preprocessor flags only for connector in debug mode

${target.driver} : LDFLAGS.debug += -LMyLibSearchPath_Driver_Debug/

Add linker flag only for connector in release mode

${target.driver} : LDFLAGS.release += -LMyLibSearchPath_Driver_Release/

Add preprocessor flags only for server in release mode

${target.server} : LDFLAGS.release += -LMyLibSearchPath_Server_Release/

Add a CXXFLAG for both connector and server in all config modes

${target} : CXXFLAGS += -Weffc++

Add a CXXFLAG for both connector and server only in debug mode

${target} : CPPFLAGS += -DMY_COMMON_MACRO

Add common search path

${target} : CPPFLAGS += $(addprefix -I, $. ${drvsrcdirs} $./MyIncludeDir1 $./MyIncludeDir2)

#...

Comparing 10.0 and 10.1 variables:

The 10.1 variables involved in this step corresponds to variables in `[DriverFolder]/Source/Makefile_FLAGS.mak` in 10.0:

Description	Name in 10.1	Name in 10.0
list of header search paths	CPPFLAGS	COMMON_CFLAGS
preprocessor flags	CPPFLAGS	COMMON_CFLAGS
compiler flags	CFLAGS, CXXFLAGS	CFLAGS CXXFLAGSCFLAGS
linker flags	LDFLAGS	BIN_LDFLAGS(_DEBUG) SO_LDFLAGS(_DEBUG)

In 10.0, compiler and preprocessor flags that are common to all config mode (release/debug) and target type (connector/server) are added to `COMMON_CFLAGS` in `[DriverFolder]/Source/Makefile_FLAGS.mak`. As well, `COMMON_CFLAGS` are always added into the implicit `CLFAGS` variable. Target-type or config-mode specific flags are conditionally appended to CFLAGS using `if-else` blocks.

Also in 10.0, linker flags that are common to all config mode and target type are added to `COMMON_LDFLAGS`. There are four other variables that represent the different combinations of target-type and config-mode specific flags: `BIN_LDFLAGS`, `BIN_LDFLAGS_DEBUG`, `SO_LDFLAGS` and `SO_LDFLAGS_DEBUG` . In 10.1, the target-type specificity is represented as target-specific rules such as `${target.driver}: LDFLAGS +=...`, and the config-mode specificity is represented with a release or debug suffix, such as `${target.server}: LDFLAGS.release +=....`

Example

This example shows a 10.0 `[DriverFolder]/Source/Makefile_FLAGS.mak` file that is roughly equivalent to the 10.1 GNU makefile example above.

```
# SDK 10.0 [DriverFolder]/Source/Makefile_FLAGS.mak

### Common compiler and preprocessor flags

COMMON_CFLAGS = $(DMFLAGS) \

-I./MySourceDir1 \

-I./MySourceDir2 \

-I./MyIncludeDir1 \

-I./MyIncludeDir2 \

-DMY_COMMON_MACRO \

-Weffc++

ifeq ($(BUILDSERVER),exe)

### Add conditional preprocessor flags for server

CFLAGS = $(COMMON_CFLAGS)

else

CFLAGS = $(COMMON_CFLAGS) -DMY_DRIVER_MACRO
```

```
endif
### Define the common linker flags
COMMON_LDFLAGS = ...
#...
ifeq ($(BUILDSERVER),exe)
### Config-mode specific linker flags for server
BIN_LDFLAGS = $(COMMON_LDFLAGS) -LMyLibSearchPath_Server_Release/
BIN_LDFLAGS_DEBUG = $(COMMON_LDFLAGS)
else
### Config-mode specific linker flags for connector
SO_LDFLAGS = $(COMMON_LDFLAGS) -LMyLibSearchPath_Driver_Release/
SO_LDFLAGS_DEBUG = $(COMMON_LDFLAGS) -LMyLibSearchPath_Driver_Debug/
```

Example Customized Makefile

This example shows a customized GNUmakefile that incorporates the modifications described in this section. All modifications are preceded with a ### comment line.

```
# Makefile for MyCustomDSII
#---------- Target Definition
buildtype = $(if ${BUILDSERVER},server,connector)
target = ${target.${buildtype}}
### Change the product names
target.driver = libMyCustomDSII${BITS}.${SO}
target.server = MyCustomDSIIServer${BITS}
### Specify directories to all folders that contain source files for this connector
drvsrcdirs = $./MySourceDir1 $./MySourceDir2
#----------
.DEFAULT_GOAL := install
clean += ${target.driver} ${target.server}
bin.install : ${target}
### Change default install location
DESTDIR.bin = $./../MyDirectory/${MODE}${BITS}
#---------- Target dependencies.
### Add source file specific to "connector"
${target.driver} : MyDriverSpecificSource1.o
```

```
### Add a flag only for connector
${target.driver} : CPPFLAGS += -DMY_DRIVER_MACRO
### Add a flag only for connector in debug
${target.driver} : CPPFLAGS.debug -DMY_DRIVER_DEBUG_MACRO
### Add preprocessor flags only for connector in debug mode
${target.driver} : LDFLAGS.debug += -LMyLibSearchPath_Driver_Debug/
### Add linker flag only for connector in release mode
${target.driver} : LDFLAGS.release += -LMyLibSearchPath_Driver_Release/
${target.driver} : LDLIBS += $(call Mutual, ${CORESDK.a} ${SQLENGINE.a} ${ODBCSDK.a})
${target.driver} : LDFLAGS += $(call LD.soname,$@)
${target.driver} : LDFLAGS += ${LD.exports}
### Add source file specific to "server"
${target.server} : MyServerSpecificSource1.o
### Add preprocessor flags only for server in release mode
${target.server} : LDFLAGS.release += -LMyLibSearchPath_Server_Release/
### Add preprocessor flags only for server in debug mode
${target.server} : LDFLAGS.release += -LMyLibSearchPath_Server_Debug/
### Add a third party library only for server
${target.server} : LDLIBS += -mythirdpartylib
### Add release-specific flags linker flag
${target.driver} : LDFLAGS.release += -lmy_release_lib
${target.server} : CPPFLAGS += ${SERVERSDK_CPPFLAGS}
${target.server} : LDLIBS += $(call Mutual, ${CORESDK.a} ${SQLENGINE.a} ${SERVERSDK.a})
${target.server} : LDLIBS += ${OPENSSL_LDLIBS}
${target.server} :; ${LINK.o} -o $@ $^ ${LDLIBS}
### Modify source file list shared by both "connector" and "server"
${target} : MyCommonSource1.o MyCommonSource2.o MyCommonSource3.o
### Add a CXXFLAG for both connector and server in all config modes
${target} : CXXFLAGS += -Weffc++
### Add a CXXFLAG for both connector and server only in debug mode
${target} : CPPFLAGS.debug += -DMY_COMMON_DEBUG_MACRO
### Remove some default search paths and add user search paths
${target} : CPPFLAGS += $(addprefix -I, $. ${drvsrcdirs} $./MyIncludeDir1 $./MyIncludeDir2)
```

```
${target} : CPPFLAGS += ${CORESDK_CPPFLAGS} ${SQLENGINE_CPPFLAGS} ${EXPAT_FLAGS}

${target} : LDLIBS += ${ICU_LDLIBS}

#--- Define search paths

vpath %.cpp ${drvsrcdirs}

vpath %.mm ${drvsrcdirs}
```

C++ on Windows

This section explains the different settings that are available on the Project Properties page in Microsoft Visual Studio. For a full listing of all compiler options, see the Microsoft MSDN documentation.

You can use the sample projects from the 5 Day Guides as an example of how to build your own custom connector. For a step-by-step example on how to build the sample projects, see the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

Build as an ODBC Connector (a DLL) for Local Connections

The sample connectors discussed in the *Build a C++ Connector in 5 Days* documents are set up to build as Windows DLLs. To build your own connector as a Windows DLL, use the following settings:

1. Set configuration type to Dynamic Library (.dll):

Select **Configuration Properties** -> **General** -> **Configuration Type**.

2. Link against `SimbaODBC.lib`. Choose the correct version of the library for release/debug, and whether MTDLL is used or not.

Select **Configuration Properties** -> **Linker** -> **Input** -> **Additional Dependencies**.

3. Set the module definition file to `Exports.def` included in all sample connectors:

Select **Configuration Properties** -> **Linker** -> **Input Module Definition File**.

4. Set the output file to a DLL name:

Select **Configuration Properties** -> **Linker** -> **General** -> **Output File**.

5. Include the DSI and Support include paths:

Select **Configuration Properties** -> **C/C++** -> **General** -> **Additional Include Directories**:

- `$(SIMBAENGINE_DIR)\Include\DSI`
- `$(SIMBAENGINE_DIR)\Include\DSI\Client`
- `$(SIMBAENGINE_DIR)\Include\Support`
- `$(SIMBAENGINE_DIR)\Include\Support\Exceptions`
- `$(SIMBAENGINE_DIR)\Include\Support\TypedDataWrapper`

Note:

If your custom connector uses the SQL Engine, see "Build with the SQL Engine" below for additional settings.

Build as a SimbaServer (an EXE) for Remote Connections

To build a connector as a stand-alone SimbaServer executable, use the following settings:

1. Set configuration type to Application (.exe):

Select **Configuration Properties** -> **General** -> **Configuration Type**

2. Link against `SimbaServer.lib`. Choose the correct version of the library for release/debug, and whether MTDLL is used or not.

   Select **Configuration Properties** -> **Linker** -> **Input** -> **Additional Dependencies**.

   For more information on these settings, see "Run-time library options" on the next page.

3. Unset the module definition file:

Select **Configuration Properties** -> **Linker** -> **Input**-> **Module Definition File**.

4. Set output file to an EXE name.

Select **Configuration Properties**-> **Linker** -> **General Output File**.

5. Include the DSI and Support include paths:

Select **Configuration Properties** -> **C/C++** -> **General** -> **Additional Include Directories**:

- `$(SIMBAENGINE_DIR)\Include\DSI`
- `$(SIMBAENGINE_DIR)\Include\DSI\Client`
- `$(SIMBAENGINE_DIR)\Include\Support`
- `$(SIMBAENGINE_DIR)\Include\Support\Exceptions`
- `$(SIMBAENGINE_DIR)\Include\Support\TypedDataWrapper`

Note:

If your custom connector uses the SQL Engine, see "Build with the SQL Engine" below for additional settings.

Build with the SQL Engine

If your custom connector uses the SQL engine, use the following setting in addition to those described in "Build as an ODBC Connector (a DLL) for Local Connections" on the previous page or "Build as a SimbaServer (an EXE) for Remote Connections" above.

1. Link against SimbaEngine.lib:

   Select **Configuration Properties** -> **Linker** -> **Input** -> **Additional Dependencies**.

2. Include the SQLEngine include paths:

   Select **Configuration Properties** -> **C/C++** -> **General** -> **Additional Include Directories**:

   - `$(SIMBAENGINE_DIR)\Include\SQLEngine`
   - `$(SIMBAENGINE_DIR)\Include\SQLEngine\AETree`
   - `$(SIMBAENGINE_DIR)\Include\SQLEngine\DSIExt`

Run-time library options

Each Simba library file has a Debug, Debug_MTDLL, Release and Release_MTDLL version. You an choose to link against any of these versions. In order to successfully link against your chosen version of the library, your project settings must match some of the settings used to build the library:

Debug

The `Debug` version of the Simba libraries are the debug version that uses a statically linked C++ runtime. To use this version of the library:

1. In **Configuration Properties** -> **C/C++**-> **Preprocessor** -> **Preprocessor Definitions**, include `_DEBUG`.
2. In **Configuration Properties** -> **C/C++** -> **Code Generation** -> **Runtime Library**, select `Multi-threaded Debug (/MTd)`.

Debug_MTDLL

The `Debug_MTDLL` version of the Simba libraries are the debug version that uses a dynamically linked C++ runtime. To use this version of the library:

1. In **Configuration Properties** -> **C/C++**-> **Preprocessor** -> **Preprocessor Definitions**, include `_DEBUG`.
2. In **Configuration Properties** -> **C/C++** -> **Code Generation** -> **Runtime Library**, select `Multi-threaded Debug DLL (/MDd)`.

Release

The `Release` version of the Simba libraries are the release version that uses a statically linked C++ runtime. To use this version of the library:

1. In **Configuration Properties** -> **C/C++**-> **Preprocessor** -> **Preprocessor Definitions**, include `_NDEBUG`.
2. In **Configuration Properties** -> **C/C++** -> **Code Generation** -> **Runtime Library**, select `Multi-threaded(/MT)`.

Release_MTDLL

The `Release_MTDLL` version of the Simba libraries are the release version that uses a dynamically linked C++ runtime. To use this version of the library:

1. In **Configuration Properties** -> **C/C++**-> **Preprocessor** -> **Preprocessor Definitions**, include `_NDEBUG`.

2. In **Configuration Properties** -> **C/C++** -> **Code Generation** -> **Runtime Library**, select `Multi-threaded DLL (/MD)`.

Character Set

In the Visual Studio “Configuration Properties” for your DSII project, on the “General” property page, ensure that the “Character Set” property is set to “Use Unicode Character Set”. This is the default setting used in the sample connector projects.

C# on Windows

This section explains the different settings that are available on the Project Properties page in Microsoft Visual Studio. For a full listing of all compiler options, see the Microsoft MSDN documentation.

As of Simba SDK 10.2.1, the Simba .NET components are packaged as NuGet (`.nupkg`) files. You should configure your NuGet environment to add the Simba SDK as a package source using this directory: *`[INSTALL_DIRECTORY]`*`\Bin\Release`.

In this section, anything referring to referencing the Simba.DotNetDSI, Simba.DotNetDSIExt, and Simba.ADO.NET assemblies can be interpreted as referencing the corresponding NuGet packages. When building for .NET Core or .NET Standard, it is strongly encouraged to only use the NuGet packages instead of directly referencing the assemblies.

Most people use the C# SDK to build an ADO.NET provider, but you can also use the C# SDK to write your DSII and build the project as an ODBC connector. The connector can be built to support either local or remote connections, with or without the SQL Engine.

You can use the sample projects from the 5 Day Guides as an example of how to build your own custom connector. For a step-by-step example on how to build the sample projects, see the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

Most of the settings described in the section "C++ on Windows" on page 137 also apply to C#, but building C# uses a different project for the .Net DSII code.

DotNetDSI and DSII

When writing a C# DSII, you must create a new Visual Studio project for a managed C# class library that does not include any of the settings described for a C++ DSII. Add the following to the project:

- If the DSII uses the Simba SQLEngine, add the Simba.DotNetDSI and Simba.DotNetDSIExt assemblies.
- If the DSII does not use the Simba SQLEngine, add the Simba.DotNetDSI assembly.

The base classes from which you derive to code your DSII are all defined in these assemblies. These assemblies can be used for both 32-bit and 64-bit connector development.

Note:

If you are using the CLIDSI, you must match the bitness of your compiled C# DLL to the bitness of the CLIDSI library that is being used.

When using your provider, server, or ODBC connector, you must register this assembly in the Global Assembly Cache (GAC) of Windows.

Simba.NET

In order to build a pure C# ADO.NET provider, you only need your DotNet DSII project and the Simba.DotNetDSI and Simba.ADO.NET assemblies.

Note:

When building a pure C# ADO.NET provider, you cannot use the Simba SQLEngine. To support this deployment scenario, use SimbaServer and the ODBC client.

When using Simba.NET to create an ADO.NET provider, you must extend the following additional abstract classes.

- `SCommand`
- `SCommandBuilder`
- `SConnection`
- `SConnectionStringBuilder`
- `SDataAdapter`
- `SFactory`
- `SParameter`

These are part of the Simba.ADO.NET assembly, not the normal DotNet DSI.

You must extend the `SConnectionStringBuilder` subclass and add any additional properties that are needed to establish a connection to your provider. For example, see the Simba DotNetUltralight sample described in the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

The rest of the classes that you extend do not typically need to be modified.

When using your provider, the Simba.DotNetDSI and Simba.ADO.NET assemblies should be registered in the Global Assembly Cache (GAC) of Windows.

Build as an ODBC Connector (a DLL) for local connections

This section explains how to build an ODBC connector for local connections, with or without the Simba SQLEngine. In addition to the DotNet DSII project, you must create a CLIDSI project that provides the bridge between the native C++ Simba ODBC libraries and your CustomerDotNetDSII.dll assembly:

1. Follow the instructions in "Build with the SQL Engine" on page 138 , including all the specified libraries.
2. Include the library `CLIDSI_$(ConfigurationName).lib`.

This library forms the bridge between the unmanaged and managed DSI classes.

3. Enable Common Language Runtime Support (/clr) by selecting the following option:

Configuration Properties -> General -> Common Language Runtime Support

4. Implement the factory function that constructs your `IDriver` object:

   Simba::DotNetDSI::IDriver^ Simba::CLIDSI::LoadDriver()

   {

   return gcnew CustomerDotNetDSII::CustomerDSIIDriver();

   }

The output of this project is your custom connector DLL, for example `CustomerCLIDSIDriver.dll`. This is the actual ODBC connector which will load your `CustomerDotNetDSII.dll` assembly.

Build as a SimbaServer (an EXE) for Remote Connections

This section explains how to build an ODBC connector for remote connections, with or without the Simba SQLEngine.

In addition to the DotNet DSII project, you must create a CLIDSI project that provides the bridge between the native C++ Simba ODBC libraries and your CustomerDotNetDSII.dll assembly:

1. Follow the instructions in "Build with the SQL Engine" on page 138, including all the specified libraries.
2. Include the library `CLIDSI_$(ConfigurationName).lib`.

This library forms the bridge between the unmanaged and managed DSI classes.

3. Enable Common Language Runtime Support (/clr) by selecting the following option:

Configuration Properties -> **General** -> **Common Language Runtime Support**

4. Implement the factory function that constructs your `IDriver` object:

   Simba::DotNetDSI::IDriver^ Simba::CLIDSI::LoadDriver()

   {

   return gcnew CustomerDotNetDSII::CustomerDSIIDriver();

   }

The output of this project is your custom connector DLL, for example `CustomerCLIDSIDriver.dll`. This is the actual ODBC connector which will load your `CustomerDotNetDSII.dll` assembly.

C# on Linux, Unix, and macOS

Simba.NET may be used to build pure C# ADO.NET providers for anywhere that .NET Core is available. All of the above in the Simba.NET section still applies.

CLIDSI cannot be used to build ODBC connectors for non-Windows platforms.

Java on Windows

This section explains how to build a connector written in Java on Windows platforms.

You can use the sample projects from the 5 Day Guides as an example of how to build your own custom connector. For a step-by-step example on how to build the sample projects, see the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

Options for Writing a Connector in Java

As explained in "Implementation Options " on page 12, you can write a custom ODBC or JDBC connector in Java using the following methods:

- Use Java to write a DSII for an ODBC connector, and connect it to the C++ SDK using a JNI component.

This option can be implemented with or without the C++ Simba SQLEngine.

- Use Java to write a DSII for an JDBC connector.

This option can be implemented with or without the Java Simba SQLEngine.

The compilation instructions for these two methods are described below.

Build a Pure-Java JDBC Connector

This type of connector can optionally use the Java SQL Engine. You can use it with or without the Java Simba SQLEngine.

Building a JDBC Connector for SQL-Capable Data Stores

The following steps describe how to build a pure-Java JDBC connector that does not use the SQL Engine:

1. Ensure that the SimbaJDBC JAR file, located at [INSTALL_DIRECTORY]\DataAccessComponents\Lib, is in the classpath.
2. Create your connector DSII JAR file.
3. No additional libraries need to be linked.

The JavaUltraLight sample connector shows how to implement and build this type of connector.

Building with the Java Simba SQLEngine

To build a pure-Java JDBC connector that uses the Java SQL engine, follow the steps in "Building a JDBC Connector for SQL-Capable Data Stores" above above, and also include the `SimbaSQLEngine.jar` in your build process.

The JavaQuickJson sample connector shows how to implement and build this type of connector. In this sample connector, the ANT build script packages the pre-compiled files with those of the DSII.

Build Java DSII for an ODBC Connector

This type of connector uses a JNI bridge to connect to the C++ API components. You can use it with or without the C++ Simba SQLEngine.

Building as an ODBC Connector (a DLL) for Local Connections

The following steps describe how to build an ODBC connector that doesn't use the SQL Engine and is not built for client-server deployments:

1. Include the settings for C++ connectors described in "C++ on Windows" on page 137.
2. Include the additional directory for the JVM library that is under `$(JAVA_HOME)\lib`:

   Configuration Properties -> **Linker** -> **General** -> **Additional Library Directories**

   Note:

   JAVA_HOME is an environment variable that should refer to the 32-bit Java installation directory when building the 32-bit ODBC connector or the 64-bit Java installation directory when building the 64-bit ODBC connector.
3. Link against `SimbaJNIDSI_$(ConfigurationName).lib` and `jvm.lib`:

   Configuration Properties -> **Linker** -> **Input** -> **Additional Dependencies**

   Set **ConfigurationName** to one of the following values: `Debug`, `Debug_MTDLL`, `Release` or `Release_MTDLL`. For information on these options, see "Run-time library options" on page 139.
4. Include the general and Java include paths:

   Configuration Properties -> **C/C++** -> **General** -> **Additional Include Directories**:

   - `$(SIMBAENGINE_DIR)\Include\JNIDSI`
   - `$(JAVA_HOME)\include`
   - `$(JAVA_HOME)\include\win32`

The sample connectors discussed in the document *Build a Java ODBC Connector in 5 Days* are configured to build as Windows DLL's.

Building as a SimbaServer (an EXE) for Remote Connection

To build a connector as a stand-alone SimbaServer executable, follow the steps in "Building as an ODBC Connector (a DLL) for Local Connections" above with the additional C++ settings described in "Build as a SimbaServer (an EXE) for Remote Connections" on page 138.

Building with the C++ Simba SQLEngine

To build a Java ODBC connector that uses the C++ SQL engine, follow the steps in "Build as an ODBC Connector (a DLL) for Local Connections" on page 137 above with the additional SQL Engine settings described in "Build with the SQL Engine" on page 138.

The JavaQuickstart sample connector shows how to implement and build this type of connector.

C++ on Linux, Unix, and macOS

The Simba SDK include a sample makefile with each of the sample connector projects. You can use this makefile to build the sample connector, then use it as a template for creating a makefile for your custom connector.

Note:

We recommend that you use the script `mk.sh` in the `Source` directory of your sample connector project. It is not recommended to use the makefile directly.

Build Configurations

The sample makefiles include targets for both debug and release versions of the connectors and the SimbaServer. The output location, or file path, indicates the bitness, compiler, and platform version. For example, when the debug version of the Quickstart connector is build on a 64-bit Linux machine with the gcc compiler, the resulting shared object is located in:

.../Bin/Linux_x86_gcc/debug64/libQuickstart64.so

Default Settings in the Sample Makefile

To help you compile and build the sample connectors on a variety of machines, the sample shell script and sample makefiles automatically detect your machine's operating system, bitness, and default compiler, then use the appropriate settings to run the build. By default, the makefiles build a release version that is dynamically linked to dependencies.

You can override these default settings by specifying them in the `mk.sh` command line, or by setting them as environment variables.

Changing the version of XCode on macOS

By default, on macOS the sample makefile detects the highest available version of the XCode compiler, and uses that to build the sample connectors. If you download a different version of the Simba SDK, you must set the active developer directory to match.

Example:

Assume you have both XCode 6 and XCode 7 on your machine, and you download the XCode 6 version of the Simba SDK. The sample makefile tries to use the XCode7 compiler, but this fails. Set the environment variable DEVELOPER_DIR to configure the active developer directory:

export DEVELOPER_DIR=/Application/Xcode6.1.app

Overriding Default Settings

The following table describes the options that you can use to override the default behaviour of the sample makefiles. Multiple options are allowed, for example:

./mk.sh MODE=debug BITS=32

Option	Description
`BUILDSERVER`	Set to `1` to build a server (`.exe`) instead of a connector (`.so` or `.dylib`). By default, a connector is built. Example: ./mk.sh BUILDSERVER=1

Option	Description
`CXX`	On Solaris, specify the compiler to use. Allowed values are the name of the compiler, for example `g++`, `g++44`, or `g++59`. Example: ./mk.sh CXX=`g++59`
`SDK_PLATFORM`	Specifies the subpath to the dependencies. This value is autodetected by default, but you can override it. The path where the dependencies are installed contains architecture information and the compiler version. For example, the ICU libraries might be installed at `ThirdParty/icu/53.1/centos5/gcc4_4`. The Simba SDK autodetects this information to allow your connector to use the correct dependencies. You can override this information to specify dependencies in a different location. Example: `./mk.sh SDK_PLATFORM=Darwin/xcode7_2`
`MODE`	Set to `debug` to build the debug version of the connector. By default, the release version of the connector is built. Example: ./mk.sh MODE=debug
`BITS`	Specifies the bitness of the connector you wish to build. By default, the makefiles build a connector that matches the bitness of your operating system, but you can override this option. For example, when building on a 64-bit platform, you can use this option to specify a 32-bit connector. Allowed values are `32`, `64`, and `3264`. Use `3264` to indicate an OSX universal binary that combines 32 and 64 bit code. Example: ./mk.sh BITS=32
`ICU_STATIC`	Set to `1` to statically link to the ICU library. By default, the makefiles build a connector that dynamically links to the ICU library. Example: ./mk.sh ICU_STATIC=1

Option	Description
OPENSSL_STATIC	Set to 1 to statically link to the OpenSSL library. By default, the makefiles build a connector that dynamically links to the OpenSS library. Example: ./mk.sh ICU_STATIC=1

Build an ODBC Connector (a Shared Object) for Local Connections

This section describes the settings you can use to build your custom ODBC connector for local connections (that is, not as a SimbaServer). You can build with or without the Simba SQLEngine. This section references the core makefile for the sample connectors, ${SIMBAENGINE_DIR}/Makefiles/kit.mk.

Note:

For each of the steps below, be sure to include the correct libraries for the compiler, bitness, and release/debug configuration.

1. Set the compiler and linker to build a shared object. The exact option depends on the compiler.
2. Include the Simba Core SDK libraries, as specified by the variable CORESDK.a in the file ${SIMBAENGINE_DIR}/Makefiles/kit.mk. Be sure to include the correct libraries for the compiler, bitness, and release/debug configuration. For example:
   - libCore.a
   - libSimbaDSI.a
   - libSimbaSupport.a
3. Include the Simba ODBC libraries, as specified by the variable ODBCSDK.a. For example:
   - libSimbaODBC.a
4. If your connector uses the SQL Engine, include the SQL Engine libraries, as specified by the variable SQLENGINE.a. For example:
   - libAEProcessor.a
   - libDSIExt.a
   - libExecutor.a
   - libParser.a
5. Include the Core SDK include paths, as specified by the variable CORESDK_CPPFLAGS. For example:

- `${SIMBAENGINE_DIR}/Include/DSI`
- `${SIMBAENGINE_DIR}/Include/DSIClient`
- `${SIMBAENGINE_DIR}/Include/Support`
- `${SIMBAENGINE_DIR}/Include/Support/Exceptions`
- `${SIMBAENGINE_DIR}/Include/Support/TypedDataWrapper`
- `${SIMBAENGINE_DIR}/ThirdParty/odbcheaders`

6. Include the ICU include paths, as specified by the variable `ICU_LDLIBS`. The path and file name contain version, bitness, and release/debug information. For example:
   - `${SIMBAENGINE_DIR}hirdParty/icu/53.1.x/Linux_x86_gcc/release64/lib`
7. If your connector uses the SQL Engine, include the SQL Engine and expat include paths, as specified by the variables `SQLENGINE_CPPFLAGS` and `EXPAT_FLAGS` in the `kit.mk` file. Note that the `Expat` directory contains a version number. For example:
   - `${SIMBAENGINE_DIR}/Include/SQLEngine`
   - `${SIMBAENGINE_DIR}/Include/SQLEngine/AETree`
   - `${SIMBAENGINE_DIR}/Include/SQLEngine/DSIExt`
   - `${SIMBAENGINE_DIR}/Include/ThirdParty/Expat/2.2.0`

Build as a SimbaServer (an EXE) for Remote Connections

This section describes the settings you can use to build your custom ODBC connector as a SimbaServer. You can build with or without the Simba SQLEngine. This section references the core makefile for the sample connectors, `${SIMBAENGINE_DIR}/Makefiles/kit.mk`.

Note:

For each of the steps below, be sure to include the correct libraries for the compiler, bitness, and release/debug configuration.

1. Set the compiler and linker to build an application (.exe). The exact option depends on the compiler.
2. Include the Simba Core SDK libraries, as specified by the variable `CORESDK.a` in the file `${SIMBAENGINE_DIR}/Makefiles/kit.mk`. Be sure to include the correct libraries for the compiler, bitness, and release/debug configuration. For example:
   - `libCore.a`
   - `libSimbaDSI.a`
   - `libSimbaSupport.a`

3. Include the Simba Server libraries, as specified by the variable `SERVERSDK.a`. For example:
   - `libSimbaCSCommon.a`
   - `libSimbaServer.a`
   - `libSimbaServerMain.a`
4. If your connector uses the SQL Engine, include the SQL Engine libraries, as specified by the variable `SQLENGINE.a`. For example:
   - `libAEProcessor.a`
   - `libDSIExt.a`
   - `libExecutor.a`
   - `libParser.a`
5. Include the Core SDK include paths, as specified by the variable `CORESDK_CPPFLAGS`. For example:
   - `${SIMBAENGINE_DIR}/Include/DSI`
   - `${SIMBAENGINE_DIR}/Include/DSIClient`
   - `${SIMBAENGINE_DIR}/Include/Support`
   - `${SIMBAENGINE_DIR}/Include/Support/Exceptions`
   - `${SIMBAENGINE_DIR}/Include/Support/TypedDataWrapper`
   - `${SIMBAENGINE_DIR}/ThirdParty/odbcheaders`
6. Include the Server SDK include paths, as specified by the variable `SERVERSDK_CPPFLAGS`. For example:
   - `${SIMBAENGINE_DIR}/Include/Server`
7. Include the ICU include paths, as specified by the variable `ICU_LDLIBS`. The path and file name contain version, bitness, and release/debug information. For example:
   - `${SIMBAENGINE_DIR}/ThirdParty/icu/53.1.x/Linux_x86_gcc/release64/lib`
8. Include the Open SLL include paths, as specified by the variable `OPENSSL_LDLIBS`. The path and file name contain version, bitness, and release/debug information. For example:
   - `${SIMBAENGINE_DIR}/ThirdParty/openssl/1.1.0/Linux_x86_gcc/release64/lib`
9. If your connector uses the SQL Engine, include the SQL Engine and expat include paths, as specified by the variables `SQLENGINE_CPPFLAGS` and `EXPAT_FLAGS` in the `kit.mk` file. Note that the `Expat` directory contains a version number. For example:

- `${SIMBAENGINE_DIR}/Include/SQLEngine`
- `${SIMBAENGINE_DIR}/Include/SQLEngine/AETree`
- `${SIMBAENGINE_DIR}/Include/SQLEngine/DSIExt`
- `${SIMBAENGINE_DIR}/Include/ThirdParty/Expat/2.2.0`

Related Topics

5 Day Guides at http://www.simba.com/resources/sdk/documentation/

SimbaClientServer User Guide at http://www.simba.com/resources/sdk/documentation/

Productizing Your Connector

In order to package your custom connector as a product for end customers, you may want to finish rebranding the configuration information and error messages. You also need to include the required dependencies in the install package, and handle configuration on the customer's machine during installation.

Packaging Your Connector

This section explains which files need to be included with your custom connector package, and what configuration steps must be performed on the customer machine in order for customers to install and use your custom connector.

The requirements for local connectors are included in this section. For client-server connectors, see the *SimbaClientServer User Guide* at http://www.simba.com/resources/sdk/documentation/.

C++ On Windows

This section explains how to package connectors written in C++ and built on Windows platforms.

1. Include the connector DLL and any additional DLLs that you added.
2. Include the ICU DLLs from `[INSTALL_DIRECTORY]\DataAccessComponents\ThirdParty\icu\53.1.x\<PLATFORM>\<CONFIGURATION>\lib\`.
   - For 32-bit connectors, include `sbicudt53_32.dll`, `sbicuin53_32.dll`, and `sbicuuc53_32.dll`.
   - Or, for 64-bit connectors, include `sbicudt53_64.dll`, `sbicuin53_64.dll`, and `sbicuuc53_64.dll`.
3. Include the error message files from `[INSTALL_DIRECTORY]\DataAccessComponents\ErrorMessages`. Include the subdirectories for the languages that you want your connector to support.
4. Create the following key in the Windows registry:
   - For 32-bit connectors on 32-bit machines, or 64-bit connectors on 64-bit machines, create the key **HKEY_LOCAL_MACHINE\SOFTWARE*Simba**Quickstart*\Driver**, replacing *Simba* with your company name and *Quickstart* with your connector name.
   - For 32-bit connectors on 64-bit machines, create the key **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node*Simba**Quickstart*\connector**, replacing *Simba* with your company name and *Quickstart* with your connector name.

   Add the following entries:
   - DriverManagerEncoding = UTF-16
   - ErrorMessagesPath = <Path to the parent directory where error message files are located>

- (OPTIONAL) LogLevel = 0
- (OPTIONAL) LogPath = <Path to directory to store the log files>

5. Create the following key in the Windows registry:
   - For 32-bit connectors on 32-bit machines, or 64-bit connectors on 64-bit machines, create the key **HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI\ODBC Drivers**.
   - For 32-bit connectors on 64-bit machines, create the key **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\ODBC\ODBCINST.INI\ODBC Drivers**.

   Add the following entry:
   - <DRIVER_NAME>=Installed
6. Create the following key in the Windows registry:
   - For 32-bit connectors on 32-bit machines, or 64-bit connectors on 64-bit machines, create the key **HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBCINST.INI*QuickstartDSIIDriver***, replacing *QuickstartDSIIDriver* with the name of your connector.
   - For 32-bit connectors on 64-bit machines, create the key **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\ODBC\ODBCINST.INI\QuickstartDSIIDrivers**, replacing *QuickstartDSIIDriver* with the name of your connector.

   Add the following entries, ensuring you include the correct path for either the 32-bit or the 64-bit connector:
   - Driver=<Full path to the connector DLL>
   - Description=<Brief description of your connector>
   - Setup=<Full path to the 32-bit connector configuration dialog DLL>

For an explanation of the registry keys that are created for the sample connectors, see *Examine the Windows Registry* and *Update the Windows Registry* in the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

C++ On Linux, Unix, and macOS

This section explains how to package drivers written in C++ and built on Linux, Unix and macOS platforms.

1. Include the connector shared object and any additional shared objects that you added.
2. Include all ICU shared objects from `[INSTALL_DIRECTORY]/DataAccessComponents/ThirdParty/icu/53.1.x/<PLATFORM>/<CONFIGURATION>/lib`.
3. Include the error message files from `[INSTALL_DIRECTORY]/DataAccessComponents/ErrorMessages`. Include the subdirectories for the languages that you want your connector to support.

4. Add the following entries to your connector's `.ini` configuration file.
   - `DriverManagerEncoding=UTF-16` (or UTF-32, depending on the driver manager being used)
   - `ErrorMessagesPath=<Path to the directory where error message file is located>`
   - `ODBCInstLib=<Full path to the Driver Manager's ODBCInst library>`
   - `(OPTIONAL) LogLevel=0`
   - `(OPTIONAL) LogPath=<Path to directory to store the log files>`
5. Add the following entries to `.odbcinst.ini`:
   - `<DRIVER_NAME>=Installed`
   - `[<DRIVER_NAME>]`
   - `Driver=<Full path to the connector shared library>`
   - `Description=<Brief description of your connector>`

For an explanation of the configuration files that are created for the sample connectors, see *Configure the Connector and Data Source* and *Configure Your Custom Connector and Data Source* in the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

C# On Windows

This section explains how to package connectors written in C#.

Packaging Connectors Built With Simba.NET

This section explains how to package connectors written in C# with the Simba.NET component.

1. Include the C# connector DLL.
2. Include `Simba.DotNetDSI.dll` and `Simba.ADO.Net.dll` from `[INSTALL_DIRECTORY]\Bin\Win`.
3. Using `gacutil.exe`, install the following DLLs to the Global Assembly Cache (GAC) on the target machine:

- `Simba.DotNetDSI.dll`
- `Simba.ADO.Net.dll`
- `Driver's C# DLL`

The DLLs can be installed using the following commands:

- `gacutil.exe /i Simba.DotNetDSI.dll`
- `gacutil.exe /i Simba.ADO.Net.dll`
- `gacutil.exe /i YourDriver.dll`

If you need to reinstall a DLL to the GAC, you have to uninstall it first using the following command:

- `gacutil.exe /u Simba.DotNetDSI`

  Note:

  The .dll extension is removed from the name when uninstalling a DLL from GAC.

Packaging Connectors Built With Simba.NET using .NET Core

- Include the entire contents of the output directory, except `.pdb` files.
- This will include your C# connector `.dll`, `Simba.DotNetDSI.dll`, `Simba.ADO.NET.dll`, and all other dependency `.dll` files.
- Unlike .NET Framework providers, this does not need to be installed in the GAC.

Packaging Connectors Built With Simba.NET using .NET Standard

- Providers built targeting .NET Standard should be packaged the same as .NET Core providers. However, they may also be installed in the GAC to be used by .NET Framework applications.
- Install `Simba.DotNetDSI.dll`, `Simba.ADO.NET.dll`, and `YourDriver.dll` into the GAC as described earlier.

Packaging Connectors Built with CLI DSI and Simba SQLEngine

This section explains how to package connectors written in C# with the CLI DSI component. The Simba SQLEngine component can optionally be included.

1. Include the requirements listed in the section "C++ On Windows" on page 151
2. Include connector's CLIDSI DLL in addition to the C# connector DLL.
3. Include `Simba.DotNetDSI.dll` and `Simba.DotNetDSIExt.dll` from `[INSTALL_DIRECTORY]\Bin\Win`.
4. In the registry, ensure the **Driver** entry is the full path to the C++ CLIDSI DLL, not to the C# DLL.
5. Using `gacutil.exe`, install the following DLLs to the Global Assembly Cache (GAC) on the target machine:

- `Simba.DotNetDSI.dll`
- `Simba.DotNetDSIExt.dll`
- `Driver's C# DLL`

The DLLs can be installed using the following commands:

- `gacutil.exe /i Simba.DotNetDSI.dll`
- `gacutil.exe /i Simba.DotNetDSIExt.dll`
- `gacutil.exe /i YourDriver.dll`

If you need to reinstall a DLL to the GAC, you have to uninstall it first using the following command:

- gacutil.exe /u Simba.DotNetDSI (NOTE: the .dll extension is removed from the name when uninstalling a DLL from GAC)

C# On Linux, Unix, and macOS

Packaging is the same as .NET Core in the above section.

Java Packaging on Windows

This section explains how to package connectors written in Java and built on the Windows platform.

Packaging JDBC Connectors Built With SimbaJDBC

This section explains how to package Java connectors built with the SimbaJDBC component.

1. Include the SimbaJDBC JAR file located at *[INSTALL_DIRECTORY]*`\DataAccessComponents\Lib`
2. Include the Java connector's JAR file.
3. If your connector uses the Java Simba SQLEngine, include the SimbaSQLEngine JAR file located at *[INSTALL_DIRECTORY]*`\DataAccessComponents\Lib`.

Packaging Java ODBC Connectors Built With JNI DSI and/or SQLEngine

This section explains how to package JDBC connectors built with the SimbaJDBC component.

1. Include the requirements listed in the section "C++ On Windows" on page 151
2. Include the connector's JNIDSI DLL and the Java connector JAR file.
3. In the registry, ensure the **Driver** entry is the full path to the C++ CLIDSI DLL, not the connector's Java JAR file.
4. Create the following key in the Windows registry:
   - For 32-bit connectors on 32-bit machines, or 64-bit drivers on 64-bit machines, create the key **HKEY_LOCAL_MACHINE\SOFTWARE*Simba**Quickstart*\Driver**, replacing *Simba* with

your company name and *Quickstart* with your connector name.

- For 32-bit connectors on 64-bit machines, create the key **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node*Simba**Quickstart*\Driver**, replacing *Simba* with your company name and *Quickstart* with your connector name.

Add the following entry. If multiple options exist, separate them by a "|" character. For example: `-Djava.class.path=<full path>\JavaQuickstart.jar>|-Xdebug:`

- `JNIConfig=<Java Virtual Machine (JVM) Configuration options>`

  Note:

  Setting the JAR Directory path in classpath will cause JNI to use anything present in that directory. If you want to mention only few JAR files in classpath, then separate each one with a semicolon(;).

  Example: `-Djava.class.path=<full_path>/JavaQuickstart.jar;<full_path>/empty.jar`

5. If `-Djava.class.path` is not specified in the JNIConfig, add or modify the CLASSPATH environment variable:

   - CLASSPATH=*<Full path of the connector's JAR file>*

   For example: `<full path>\JavaQuickstart.jar`

6. Add or modify the PATH environment variable to include the location of the Java executable, as well as the 64-bit or 32-bit Java Virtual machine (Depending on the bitness of JNIDSI connector).

   Note:

   For a Java Runtime Environment (JRE), the location of the JVM on 32-bit Windows is usually in <JRE Path>\bin\client while on 64-bit it is usually in <JRE Path>\bin\server.

Java Packaging On Linux, Unix, and macOS

This section explains how to package connectors written in Java and built on Linux, Unix, or macOS platforms.

Packaging JDBC Connectors Built With SimbaJDBC

This section explains how to package Java connectors built with the SimbaJDBC component.

1. Include the SimbaJDBC JAR file located at `[INSTALL_DIRECTORY]\DataAccessComponents\Lib`.
2. Include the Java connector's JAR file.
3. If your connector uses the Java Simba SQLEngine, include the SimbaSQLEngine JAR file located at `[INSTALL_DIRECTORY]/DataAccessComponents/Lib`.

Packaging Java ODBC Connectors Built With JNI DSI and/or SQLEngine

This section explains how to package Java connectors written in Java and built with the SimbaJDBC component.

1. Include the requirements listed in the section "C++ On Linux, Unix, and macOS" on page 152.
2. Include the connector's JNIDSI library and the Java connector JAR file.
3. In the .ini file, ensure the **Driver** entry is the full path to the C++ CLIDSI DLL, not the connector's Java JAR file.
4. Add the following entry to your connector's .ini configuration file. If multiple options, separate them with a "`|`" character.
   - `JNIConfig=<Java Virtual Machine (JVM) Configuration options>`

   For example, `-Djava.class.path=<full path>/JavaQuickstart.jar>|-Xdebug`

   Note:

   Setting the JAR Directory path in classpath will cause JNI to use anything present in that directory. If you want to mention only few JAR files in classpath, then separate each one with a semicolon(;).

   Example: `-Djava.class.path=<full_path>/JavaQuickstart.jar;<full_path>/empty.jar`
5. If `-Djava.class.path` is not specified in the JNIConfig, add or modify the CLASSPATH environment variable:
   - CLASSPATH=*<Full path of the connector's JAR file>*

   For example: `<full path>\JavaQuickstart.jar`
6. Add or modify the LD_LIBRARY_PATH (or equivalent) environment variable to include the location of the Java executable, as well as the 64-bit or 32-bit Java Virtual machine, depending on the bitness of JNIDSI connector.

For a Java Runtime Environment (JRE), the location of the JVM on 32-bit Unix is usually in `<JRE Path>/lib/<architecture>/client`, where *architecture* is `amd64` on 64-bit linux, or `i386` on 32-bit linux, or other values on other platforms.

Note:

The library path environment variable has the following values on the different platforms:

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

Related Topics

"Including Error Message Files" on page 80

Adding a DSN Configuration Dialog

You can add a custom dialog in the ODBC Data Source Administrator. This dialog is displayed when users click Add, Remove, or Configure. By using your custom dialog, customers can perform custom configuration of the ODBC connection without having to modify the Windows registry or by editing the `.ini` files on Unix or Linux platforms.

To create a custom DSN configuration dialog, implement the connector setup DLL API by implementing and exporting the `ConfigDSN` function:

```
BOOL INSTAPI ConfigDSN(
    HWND in_parentWindow,
    WORD in_requestType,
    LPCSTR in_driverDescription,
    LPCSTR in_attributes)
```

You can implement this in the connector shared library, or as a separate shared library.

Note:

- If you build the configuration dialog as a separate DLL, we recommend changing the extension from.dll to .cnf, as this is conventional practice.
- The QuickStart sample project provides an example of implementing `ConfigDSN` and exporting it from within the connector DLL.

To get your setup function recognized by the ODBC Data Source Administrator, you must add the **Setup** key to your connector's entry in **ODBCINST.INI** in the registry. For an example of adding the Setup key, see *C++ Packaging for Windows* in the *Simba SDK Deployment Guide*.

For more information on creating a DLL, refer to the Setup DLL API Reference and the Installer DLL API Reference Function in the Microsoft ODBC Programmer's Reference.

Rebranding Your Connector

The sample connectors are branded with a default connector name, for example *Quickstart*, and the default company name *Simba*. These names are visible to customers in the following locations:

- The Windows registry
- The `.ini` configuration files on Linux, Unix, and macOS
- The vendor name in error messages

The Simba SDK allows you to rebrand your custom connector with the connector name and company name of your choice. The 5 Day Guides at http://www.simba.com/resources/sdk/documentation/ provide detailed instructions on how to use the rebranding functionality in Windows, Linux, Unix, and macOS.

Using INI Files for Connector Configuration on Windows

On Windows platforms, ODBC connectors normally retrieve configuration information from the Windows registry. As an alternative, your connector can retrieve its connector-specific configuration information from an `.ini` file. This enables customers to deploy multiple versions of the DLL, each with a different version of the connector-specific configuration information. You can also specify that your connector to look for the `.ini` file initially, then fall back to the registry if the file cannot be found.

You can use a configuration file, for example `simba.quickstart.ini`, to specify the information that is normally retrieved from the **SOFTWARE\Simba\Quickstart\Driver** registry key.

Note:

- The registry key and the file name can be rebranded with your own company and connector name, but this example uses *simba* and *Quickstart* for simplicity.
- This feature does not include using `.ini` files for information that is stored in the **\ODBC\ODBC.INI** and **\ODBC\ODBCINST** registry keys. You still need to configure these registry keys for your custom connector.

Step 1 - Create the simba.quickstart.ini file

Create a text file that contains all the information in the connector's **HKEY_LOCAL_MACHINE\SOFTWARE\Simba\Quickstart\Driver** registry key. The file has the same format as the `simba.quickstart.ini` file on Linux and Unix platforms.

Example - simba.quickstart.ini file

[Driver]

ErrorMessagesPath=C:\Drivers\Quickstart\Maintenance\10.1\Product\ErrorMessages

UnicodeDataPath=C:\Drivers\Quickstart\Maintenance\10.1\Product\Setup

DriverLocale=en-US

DriverManagerEncoding=UTF-16

LogLevel=3

LogNamespace=

LogPath=C:\SimbaLogs

Step 2 - Update Simba::DSI::DSIDriverFactory()

In the `Simba::DSI::DSIDriverFactory()` method in the `Main_Windows.cpp` file, replace the call to `SetConfigurationBranding()` with `SetConfigurationRegistryKey()`, `SetConfigurationIniFile()`, and `SetModuleId()`. These methods must be called before any parameter from the `SimbaSettingReader` is accessed, because the configuration is loaded only once, and cannot be reloaded.

You also need to provide the module ID, using the value provided to the `DLLMain()` function that is called when the connector is loaded.

Example - DSIDriverFactory

```
//==============================
/// @brief Creates an instance of IDriver for a connector.
/// The resulting object is made available through DSIDriverSingleton::GetDSIDriver().
/// @param out_instanceID Unique identifier for the IDriver instance.
/// @return IDriver instance. (OWN)
//==============================
IDriver* Simba::DSI::DSIDriverFactory(simba_handle& out_instanceID)
{
	out_instanceID = s_quickstartModuleId;

	//Set the name of the INI file from which to load the connector-specific configuration.
	// If a file name is specified here, the SEN SDK will first try to load the connector specific
	// configuration from that INI file. If it can't find the file, it will fall
	// back to the registry, as described below.
	// You can specify a relative path for the file name. If the module ID (see below) is
	// 0, then the path is relative to the current working directory. If the module
	// ID is non-zero, the path is relative to the
	// directory where the DLL is located.

	#if defined(SERVERTARGET)
	SimbaSettingReader::SetConfigurationIniFile("simbaserver.quickstart.ini");
	#else
	SimbaSettingReader::SetConfigurationIniFile("simba.quickstart.ini");
	#endif

	// Set the module identifier provided in DLLMain().
	//This allows the SDK to determine in which
	// directory the connector DLL is located, which is used to load INI file defined
	// as relative path as described above.

	SimbaSettingReader::SetModuleId(s_quickstartModuleId);
```

```
    // Use this setting to specify the registry key that is used if the
    // connector cannot find the .ini file.
    // For example, if you use the value "Simba\Quickstart", then
    // the connector looks for the configuration information at
    // HKLM\SOFTWARE\Simba\Quickstart, or
    // HKLM\SOFTWARE\SOFTWARE\Wow6432Node\Simba\Quickstart if
    // running a 32-bit connector on 64-bit Windows).
    // If the DSII is compiled as a connector, then it will use \Driver as the final
    // value in the registry path.
    // If the DSII is compiled as a server, then it will use \Server as the final
    // value in the registry path.
    // For example, a 64-bit connector would use
    // HKLM\SOFTWARE\Simba\Quickstart\Driver to look up the registry keys such as
ErrorMessagesPath.

    SimbaSettingReader::SetConfigurationRegistryKey("Simba\\Quickstart");

    // Set the server branding for this data source. This will only be used if the DSII is compiled
    // as a server and then installed as a service.
    #if defined(SERVERTARGET) && defined(WIN32)
    SimbaSettingReader::SetServiceName("SimbaQuickstartService");
    #endif

    return new QSDriver();
}
```

Logging to Event Tracing for Windows (ETW)

By default, the Simba SDK logging functionality writes events and messages to text files. You can develop your custom ODBC or JDBC connector log events and messages to Event Tracing for Windows (ETW) instead. You can also enable it to switch between file and ETW logging at runtime.

The basic steps are as follows:

1. Define the provider GUID in your connector code
2. Use the `ETWLogger` Class in your connector code

For an example of how to implement ETW logging in the QuickStart sample OBBC connector, see "Example: Implementing ETW Logging" on page 164.

Step 1 - Define the Provider GUID in your Connector Code

When creating a manifest file to define your provider, you created a provider GUID. Add this GUID to the connector's main header file. If you have both 32 and 64-bit connectors, you need to include both GUIDs. If your connector code is used on multiple platforms, ensure the GUID is defined just for Windows platforms. For example:

```
#if defined(_WIN64)

/// The 64-bit connector specific ETW provider GUID.

const GUID PROVIDER_GUID = {0x69bacf08, 0x09d0, 0x400a, {0xab, 0xd8, 0x52, 0x06, 0xd4, 0xbd, 0x79, 0x39}};

#elif defined(WIN32)

/// The 32-bit connector specific ETW provider GUID.

const GUID PROVIDER_GUID = {0x9bbc191d, 0x1d80, 0x40d1, {0xad, 0xab, 0xe1, 0x1b, 0x97, 0x3a, 0x1e, 0x90}};

#endif
```

Step 2 - Use the ETWLogger Class in your Connector Code

Change your custom connector code to use the `ETWLogger` class instead of the `DSIFileLogger` class.

For example, in the Quickstart sample connector you would have the following `QSDriver` constructor

```
QSDriver::QSDriver() : DSIDriver(), m_driverLog(new ETWLogger(PROVIDER_GUID))

{

    ENTRANCE_LOG(m_driverLog, "Simba::Quickstart", "QSDriver", "QSDriver");

    SetDriverPropertyValues();

    ...

}
```

Further Considerations

You may want to refine the example shown in "Example: Implementing ETW Logging" on page 164 with the additional functionality described in this section.

Understanding Log Levels in Windows ETW Logging

The Simba SDK supports six different log levels for file-based logging, and four different log levels for ETW-based logging. The following table shows the mapping between ETW log level and the LogLevel setting in the Windows registry or `.ini` file.

LogLevel setting	ETW Log Level
1	1 (Fatal)
2	2 (Error)
3	3 (Warning)
4,5,6	4 (Debug, Information, Trace)

For example, if you configure **LogLevel** to `6`, then Debug, Information, and Trace logs are all logged as level 4 in the ETW logger.

Increasing the File Size

By default, the maximum size of the log files for ETW logging is 1028 KB. Subsequent events are overwritten in the file. In order to see more events in the log file, you may want to increase the maximum file size by selecting **properties** in the Event Viewer.

Set an Activity ID

By default, the activity ID for the events is set to `0`. To change this activity ID to the string of your choice, use `ETWLogger::SetActivityId()`.

Enable logging for both 32-bit and 64-bit connectors

If you plan to ship a 32-bit and a 64-bit version of your connector, you need to create a manifest file for each version. For example, create a `QuickStart32.man` and a `QuickStart64.man`.

> **Important:**
> **Important:**
>
> Be sure you create a different GUID for the 32-bit and the 64-bit manifest files. Each manifest file must have its own, unique GUID.

In the source code, define each provider GUID for the correct platform.

Example:

```
#if defined(_WIN64)

/// The 64-bit connector specific ETW provider GUID.

const GUID PROVIDER_GUID = {0x69bccf01, 0x08d0, 0x400a, {0xbb, 0xc8, 0x52, 0x06, 0xb4, 0xbd, 0x72, 0x39}};

#elif defined(WIN32)

/// The 32-bit connector specific ETW provider GUID.

const GUID PROVIDER_GUID = {0x9bbc737c, 0x1d70, 0x40d9, {0xad, 0xab, 0xe1, 0x8b, 0x17, 0x3a, 0x4e, 0x20}};
```

#endif

Allowing the user to switch between ETW and File logging

If you want allow your customers to switch between ETW logging and file logging, you can create a registry key that defines the type of logging. Then in your code, instantiate the correct logging class depending on the registry setting.

Related Topics

"Example: Implementing ETW Logging" below

Example: Implementing ETW Logging

This example shows one way that you could configure ETW logging for the QuickStart connector. The steps are:

- "Step 1 - Create a Manifest File for the QuickStart Connector" below
- "Step 2 - Create the Master Resources File" on page 167
- "Step 3 - Update the QuickStart Connector Code" on page 168
- "Step 4 - Configure ETW to Log QuickStart Events" on page 168
- "Step 5 - Generate Loggable Activity in the QuickStart Connector" on page 170

Note:

This examples helps you get started. For more details, see "Further Considerations" on page 162.

Step 1 - Create a Manifest File for the QuickStart Connector

Create a manifest file to define the QuickStart connector as an event provider, then compile the file to generate resources.

To create the manifest file:

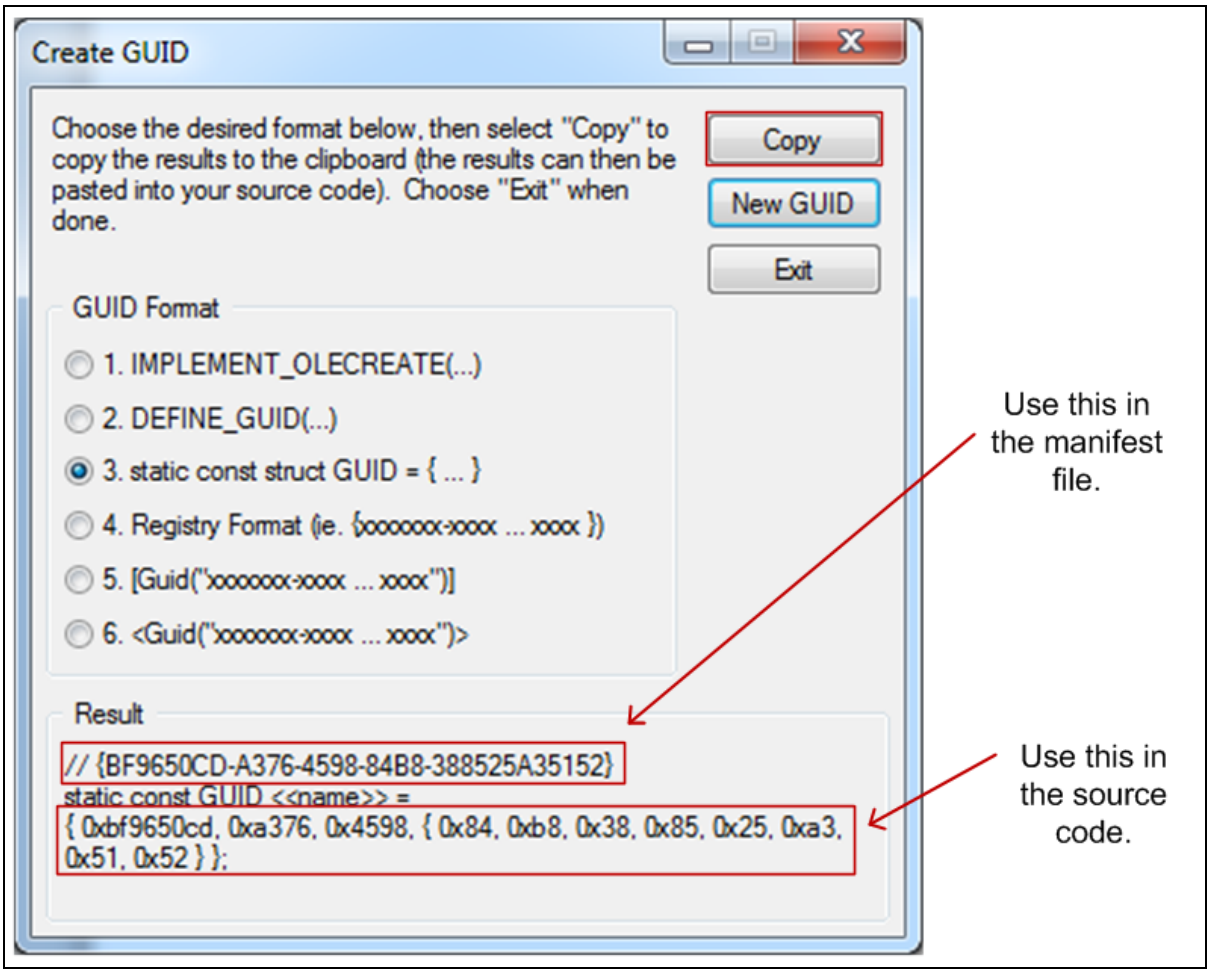
1. Copy the following XML into a text editor:

```
<?xml version="1.0" encoding="UTF-16"?> <instrumentationManifest
xsi:schemaLocation="http://schemas.microsoft.com/win/2004/08/events
eventman.xsd"
xmlns="http://schemas.microsoft.com/win/2004/08/events"
xmlns:win="http://manifests.microsoft.com/win/2004/08/windows/event
s" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```
xmlns:trace="http://schemas.microsoft.com/win/2004/08/events/trace">
 <instrumentation> <events> <provider name="DriverName" guid="
{MYGUID}" symbol="DriverName" resourceFileName="Path to connector
dll" messageFileName="Path to connector dll"> <events> <event
symbol="DebugInfoTraceEvent" value="0" version="0" channel="Admin"
level="win:Informational" template="AllEventsTemplate"
message="$(string.DriverName.event.0.message)"> </event> <event
symbol="ErrorEvent" value="1" version="0" channel="Admin"
level="win:Error" template="AllEventsTemplate"
message="$(string.DriverName.event.1.message)"> </event> <event
symbol="FatalEvent" value="2" version="0" channel="Admin"
level="win:Critical" template="AllEventsTemplate"
message="$(string.DriverName.event.2.message)"> </event> <event
symbol="WarnEvent" value="3" version="0" channel="Admin"
level="win:Warning" template="AllEventsTemplate"
message="$(string.DriverName.event.3.message)"> </event> </events>
<levels> </levels> <channels> <channel name="Admin" chid="Admin"
symbol="Admin" type="Admin" enabled="true"> </channel> </channels>
<templates> <template tid="AllEventsTemplate"> <data name="message"
inType="win:UnicodeString" outType="xs:string"> </data> </template>
</templates> </provider> </events> </instrumentation> <localization>
 <resources culture="en-US"> <stringTable> <string
id="level.Warning" value="Warning"> </string> <string
id="level.Informational" value="Information"> </string> <string
id="level.Error" value="Error"> </string> <string
id="level.Critical" value="Critical"> </string> <string
id="DriverName.event.3.message" value="%1"> </string> <string
id="DriverName.event.2.message" value="%1"> </string> <string
id="DriverName.event.1.message" value="%1"> </string> <string
id="DriverName.event.0.message" value="%1"> </string> </stringTable>
 </resources> </localization> </instrumentationManifest>
```

2. Save the file as `Quickstart.man`.

3. Replace every instance of `DriverName` with `QuickStart`.
4. Find and replace `Path to connector dll` with the complete path to your connector. Be sure to use the correct DLL for debug, platform, and bitness.
5. Generate a new GUID and use it to replace *`MYGUID`*:
   a. In Visual Studio, select **Tools** > **Create Guid**.
   b. Select option **3** then select **Copy.**



   c. Paste the first line into the manifest file, and save the second line to paste into your source code.

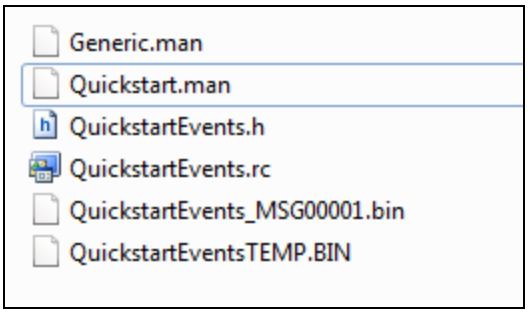
6. Save and close the `Quickstart.man` file.

To compile the manifest file:

1. Open a command prompt and navigate to the directory where your `Quickstart.man` file is stored.
2. Run the following command:

"C:\Program Files (x86)\Windows Kits\8.1\bin\x64\MC" Quickstart.man -um -z QuickstartEvents

The manifest file is compiled and the resource files are generated:



Step 2 - Create the Master Resources File

Create a master resources file to consume the resources you generated in the previous step.

To create the master resources file:

1. In the `Source/Resources` folder of your Visual Studio project, create a text file called `Master.rc`.

2. Add this file to the connector's Visual Studio project:
   a. Right click the **Resources** folder in the connector project in Visual Studio and select **Add** > **Existing Items**
   b. Browse to **Resources** folder, select the **Master.rc** file that you created, and click **Add**.

3. In a text editor, open the **Master.rc** file and `#include` all of the `.rc` files in the QuickStart project. Also include the files you generated in the previous step. Save and close the file.

   Example: Master.rc file

   #include "Dialogs.rc"

   #include "QuickstartVersion.rc"

   #include "QuickstartEvents.rc"

4. In a text editor, open the Visual Studio project and remove references to any resource files other than the `Master.rc` file.

   Example: Remove the lines shown below

   <ItemGroup>

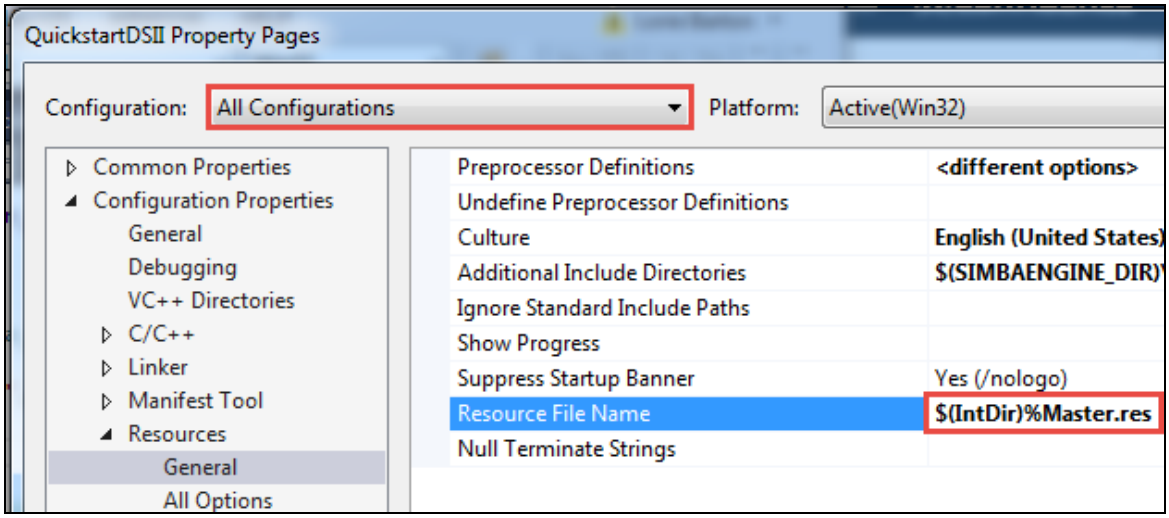
   ~~<ResourceCompile Include="Resources\Dialogs.rc" />~~

   <ResourceCompile Include="Resources\Master.rc" />

   ~~<ResourceCompile Include="Resources\QuickstartVersion.rc" />~~

   </ItemGroup>

5. Update the QuickStart connector project:
   a. In Visual Studio, right-click the QuickStart project and select **Properties**.
   b. Select **Configuration Properties** > **Resources** and select **General**.
   c. Change the **Resource File Name** field to **$(IntDir)\Master.res** .
   d. Click **Apply**.



Step 3 - Update the QuickStart Connector Code

Modify the QuickStart connector code to use the `ETWLogger` class instead of the default file logger class.

To use the ETW logger class:

1. In Visual Studio, open the QuickStart solution, then open the file **Core** > **Include** > **QSDriver.h**.
2. Define the GUID you created earlier. This will be the connector's provider GUID.

Example: In QSDriver.h

```
namespace Simba

{

namespace Quickstart

{

const GUID PROVIDER_GUID = { 0xf77f8f1e, 0xb189, 0x49ed, { 0xa3, 0xd9, 0xab, 0x72, 0x39, 0x19, 0xd2, 0x17 } };
```

3. Open the `QSDriver.cpp` file and add the following line:

   ```
   #include "ETWLogger.h"
   ```

4. In the QuickStart connector's constructor, change the existing logger to `ETWLogger`. Pass in the `PROVIDER_GUID`.

Example:

```
QSDriver::QSDriver() : DSIDriver(), m_driverLog(new ETWLogger(PROVIDER_GUID))
```

Step 4 - Configure ETW to Log QuickStart Events

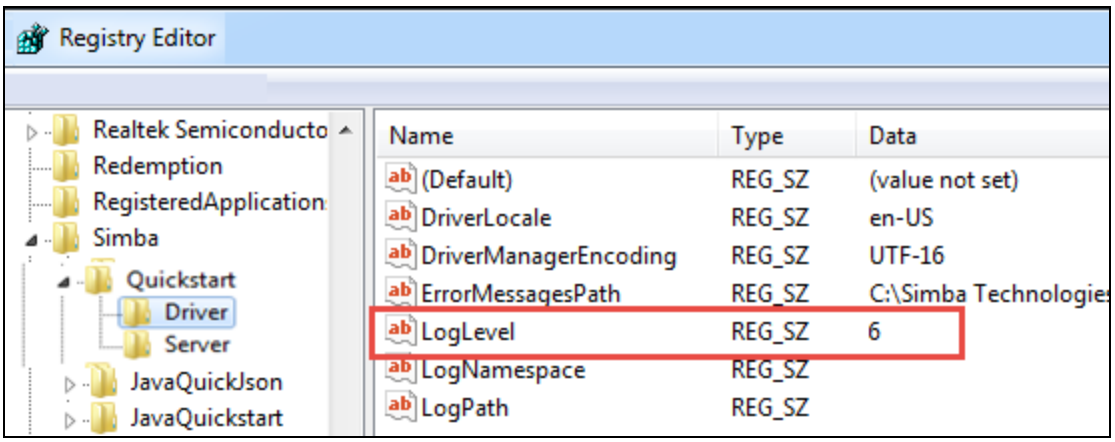
Register the connector with ETW and enable logging.

To configure ETW to log QuickStartEvents:

1. Register the QuickStart connector as an ETW provider:
   a. Open a command prompt with administrator privileges.
   b. In the directory containing the `Quickstart.man` file, type the following command:

      wevtutil im Quickstart.man /resourceFilePath:"C:\Simba Technologies\SimbaEngineSDK\10.1\Examples\Source\Quickstart\Bin\Windows_vs2013\debug32md\QuickstartDSIIODBC32.dll" /messageFilePath:"C:\Simba Technologies\SimbaEngineSDK\10.1\Examples\Source\Quickstart\Bin\Windows_vs2013\debug32md\QuickstartDSIIODBC32.dll"

2. Ensure the connector is configured for logging by setting the following registry key to `6`:>
   - Use **HKEY_LOCAL_MACHINE\SOFTWARE\Simba\Quickstart\Driver** for a 32-bit connector on a 32-bit machine or a 64-bit connector on a 64-bit machine
   - Or, use **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Simba\Quickstart\Driver\LogLevel** for a 32-bit connector on a 64-bit machine

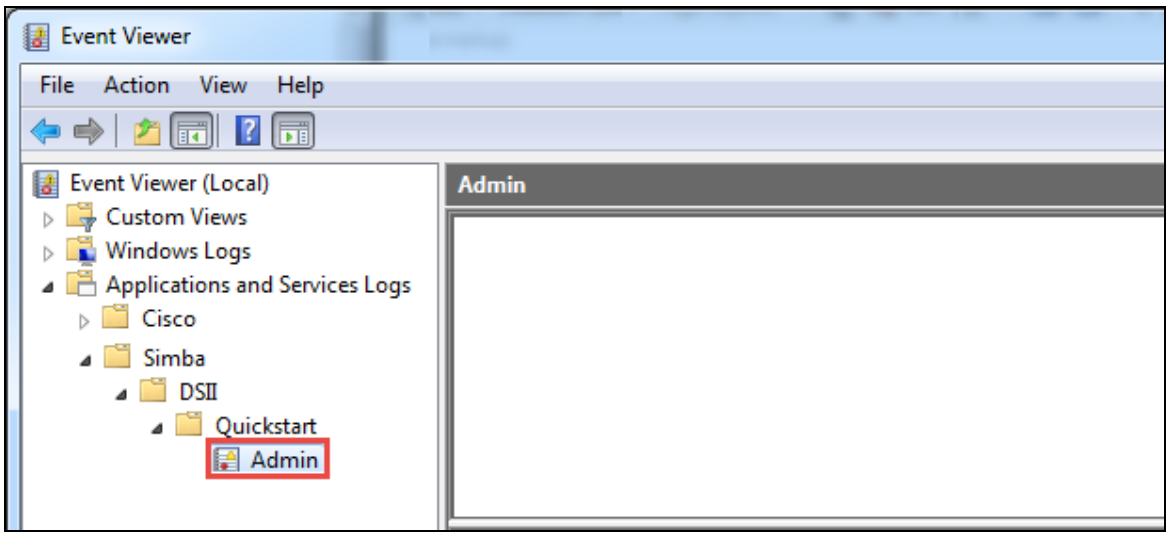


3. Open Event Viewer by typing **Event Viewer** in the Start Menu.
4. In Event Viewer, expand **Applications and Services Logs** > **Simba** > **DSII** > **Quickstart** and select **Admin**.

   Note:

   If nothing appears under Applications and Services Logs, wait a few minutes for Event Viewer to populate.

5. Right-click on **Admin** and select **Enable Log**.



Step 5 - Generate Loggable Activity in the QuickStart Connector

To see events logged in ETW, you need to configure the connector to start logging, then use it for something such as establishing a connection.

To create activity that will be logged:

1. Navigate to the folder containing the ODBC Test application, by default:

```
C:\Program Files (x86)\Microsoft Data Access SDK 2.8\Tools
```

2. Navigate to the folder that corresponds to your connector's architecture: **amd64**, **ia64** or **x86**.

   Example:

   If you built the 32-bit version of your connector on a 64-bit machine, select the **x86** version.

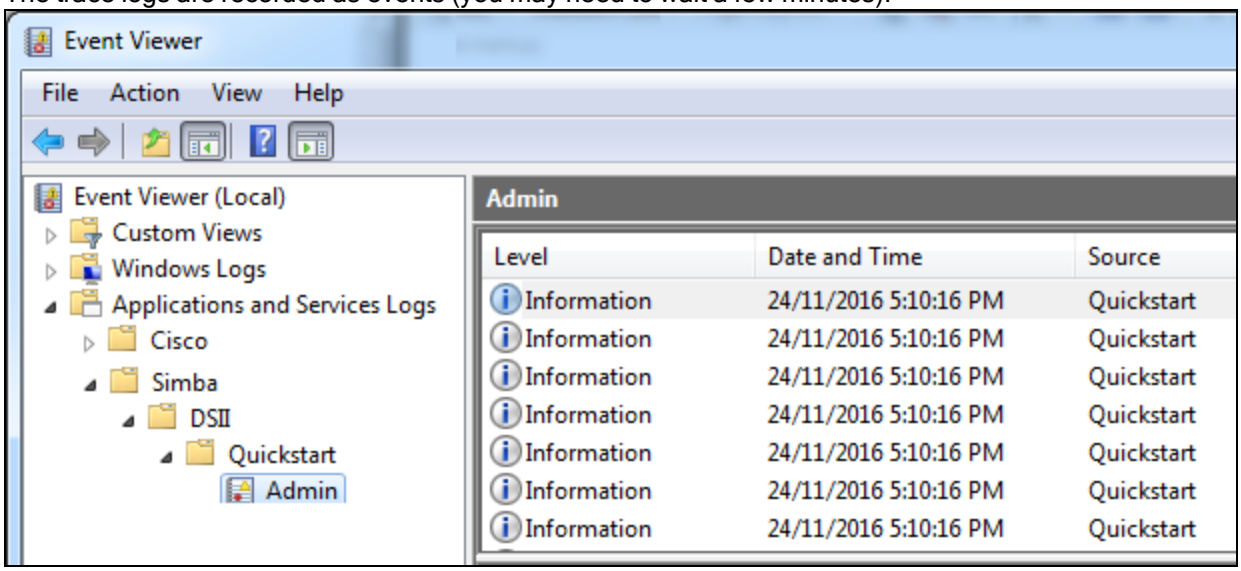
3. Click one:

- **odbcte32.exe** to launch the ANSI version
- Or, **odbct32w.exe** to launch the Unicode version.

> **Important: Important:**
>
> Make sure that you run the correct version of the ODBC Test tool for ANSI or Unicode and 32-bit or 64-bit.

4. In the ODBC Test tool, click **Conn** > **Full Connect**.
5. In Event Viewer, navigate to **Applicationsand Services Logs** > **Simba** > **DSII** > **Quickstart** and select **Admin**.

6. The trace logs are recorded as events (you may need to wait a few minutes):



Related Topics

"Logging to Event Tracing for Windows (ETW)" on page 161

For information on rebranding the Simba registry key to your own company name, see "Rebranding Your Connector" on page 158.

Testing your DSII

During the development of your connector, you may want to test its functionality using applications such as Microsoft Excel on Windows or iODBCTest on Linux. This section explains how to use different applications to test your connector. It also explains how to resolve common problems and errors messages that you may encounter at different stages of development.

Testing On Windows

This section explains how to use Microsoft Access, Microsoft Excel, and the ODBCTest tool to test your custom ODBC connector.

Testing With Microsoft Access

Running your connector against Microsoft Access is a good test to prove basic functionality. Microsoft Access uses much of the ODBC API, including many of it's edge cases.

Note:

To get the widest test coverage of the ODBC API, test your connector under Microsoft Access by loading your data as linked tables.

To Test Your Custom ODBC Connector with Microsoft Access:

1. Open Access and create a new Blank Database.
2. Select **External Tab** -> **More** -> **ODBC Database**.
3. In the Get External Data - ODBC Database dialog, select **Link to the data source by creating a linked table**.
4. In the Select Data Source dialog, select the **Machine Data Source** table and choose your DSN. Click **OK**.
5. In the Link Tables dialog, select the tables you want to link to. Click **OK**.
6. In the Select Unique Record Identifier dialog, click **OK** without choosing any specific field.
7. In the All Table panel, right click on a table name and click **Open**.

The data from the table appears in Microsoft Access.

Testing with Microsoft Excel

You can test your data source with Microsoft Excel by importing data using the Data Connection Wizard.

To Test Your Custom ODBC Connector with Microsoft Excel:

1. Open Excel and create a new blank workbook.
2. Select **Data** -> **From Other Sources** -> **From Data Connection Wizard**.
3. In the Data Connection Wizard dialog, select **ODBC DSN** and click **Next**.
4. In the ODBC data sources box, Select your DSN and click **Next**.

5. In the Table box, select a table and click **Next** and then click **Finish.**
6. In the Import Data dialog, select **Table** for how you want to view the data in the workbook and click **OK**.

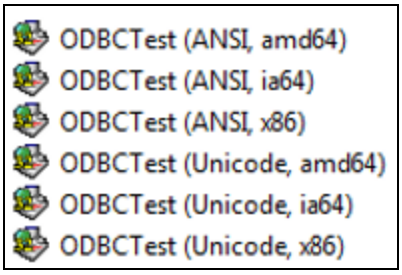
The data from the table appears in your Excel workbook.

Testing with ODBCTest

ODBCTest is a test application provided by Microsoft as part of the Microsoft Data Access Components (MDAC) SDK and the Platform SDK. For more information about MDAC, see "What is MDAC?" on page 26.

This application allows you to manually execute any SQL query. You can also use it to directly call any method in the ODBC API. The full ODBC API is exposed through the ODBCTest menus, allowing you to walk through each step of an ODBC API call and viewing the results in real time.

The MDAC installation includes both ANSI and Unicode-enabled versions of ODBC Test, for both 32-bit and 64-bit systems. The versions are clearly marked in the Programs menu:



Note:

- Select the correct version of MDAC for the bitness of your connector and ANSI or Unicode requirements.
- On 64-bit Windows, ensure that your DSNs are configured correctly.

Running your connector under the debugger with ODBCTest configured as the launch application is an excellent way to test. You can set breakpoints in your DSII, and break into them as various ODBC calls trigger corresponding DSII calls. You can break at every DSII API call, and step through the execution of each of your DSII methods to track down problems with precision.

For more information on using Visual Studio to debug into your custom ODBC connector with ODBCTest, see the section *Debug the Custom ODBC Connector* in the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

To Test Your Custom ODBC Connector with ODBCTest:

Before running this test, ensure you have already configured a DSN for your connector. For more information on creating a DSN in the Windows registry for your custom ODBC connector, see *Update the Windows Registry* in the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

1. Start the ODBCTest application, ensuring you run the correct version for ANSI or Unicode and 32-bit or 64-bit.
2. To configure the application to use ODBC 3.52 menus, select **Tools** -> **Options** menu.

3. Select the **ODBC Menu Version** -> **ODBC 3.x**.
4. Create a connection to your connector using the appropriate DSN you already configured.
5. Select **Connect** -> **Full Connect**.

This option allocates the environment and connection handles, then opens the connection.

6. Locate your DSN in the data source list.
7. Select **OK**. A new window appears with the message "Successfully connected to DSN '<your connector name>". The top half of the window allows you to enter SQL queries to be executed. The bottom half displays the results.

   Note:

   If you see the error "SQLDriverConnect returned: SQL_ERROR=-1", use the following tips for troubleshooting. This error usually occurs because the Windows Driver Manager cannot find or load the requested connector's DLL. Check the following:

   - Does your DSN exist in the registry both as a registry key in **ODBC.INI** and as a value in **ODBC.INI\ODBC Data Sources**?
   - Does your connector exist in the registry both as a registry key under **ODBCINST.INI** and as a value in **ODBCINST.INI\ODBC Drivers**?
   - Does your DSN have a **Driver** entry?
   - At the path specified in the DSN's **Driver** entry, does the specified DLL exist?

8. Enter a simple SQL query by selecting **Statement** -> **SQLExecDirect**. Select **OK** on the resulting dialog.
9. From the Results menu, select **GetDataAll**.
10. Review the results.
11. Select **Catalog** -> **SQLTables**. Select **OK** on the resulting dialog.

SQL Catalog functions work only if you have implemented the appropriate MetadataSources.

12. Select **Results** -> **GetDataAll**.
13. Review the results

Related Topics

Debug the Custom ODBC Connector in the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/

Update the Windows Registry in the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/

Testing On Linux, Unix, and MacOS

This section explains how you can test your custom ODBC connector in Linux, Unix, and macOS platforms.

iODBCTest and iODBCTestW

The utilities iodbctest and iodbctestw are included with the iODBC driver manager installation. You can use one these utilities to establish a test connection with your connector and your DSN. Use iodbctest to test how your connector works with an ANSI application, and use iodbctestw to test how your connector works with a Unicode application.

For more information on how to use this utility, see www.iodbc.org. For an example of how to use iODBCTest, see the section *Connect to the Data Source* in the Linux or macOS version of the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/.

To Test Your Custom ODBC Connector with iODBCTest or iODBCTestW:

1. Use the following command to run `iodbctest` or `iodbctestw`:

   `./iodbctest`

   Or, `./iodbctestw`

   Note:

   There are 32-bit and 64-bit installations of the iODBC driver manager available. If you have only one version of the driver manager installed, you will have the appropriate version of iodbctest (or iodbctestw). However, if you have both 32-bit and 64-bit versions installed, you will need to ensure that you are running the version from the correct installation directory.

2. The program will ask you to enter an ODBC connect string. Type `?` if you do not remember the name of your DSN. Your ODBC connect string has the following format:

   `DSN=<your_DSN_name>;UID=<user_id> (if applicable);PWD=<your password>` (if applicable)

3. If you have successfully connected, the prompt `SQL>` appears.

4. Test out some simple SELECT queries to see if your data is being retrieved properly from your data source.

UnixODBC

iSql is a utility that is included with the UnixODBC driver manager installation. You can use this utility to test a connection with your connector and your DSN.

For more information on how to use this utility, see www.iodbc.org.

1. Run iSql:

   ./isql *<DSN> <UID (if applicable)> <PWD (if applicable)> <options (if applicable)>*

2. If you have successfully connected, the prompt `SQL>` appears.

3. Test out some simple SELECT queries to see if your data is being retrieved properly from your data source.

Related Topics

Connect to the Data Source in the Linux or macOS version of the 5 Day Guides at http://www.simba.com/resources/sdk/documentation/

Driver Manager Encodings on Linux, Unix, and MacOS

On Linux, Unix, and macOS platforms, it is possible to specify that an application use a particular driver manager. You may need to configure a connector to work with an application, depending on which driver manager has been linked to the application. The connector configuration file can set the `DriverManagerEncoding` setting to indicate what type of Unicode is being passed to the connector from the driver manager.

The following table outlines the Unicode setting to use:

Platform	Bitness	iODBC	UnixODBC
Linux	32	UTF-32	UTF-16
Linux	64	UTF-32	UTF-16
Linux Itanium	64	UTF-32	UTF-16
AIX (PowerPC)	32	UTF-16	UTF-16
AIX (PowerPC)	64	UTF-32	UTF-16
macOS	32	UTF-32	
macOS	64	UTF-32	
HP-UX (Itanium)	32	UTF-32	UTF-16
HP-UX (Itanium)	64	UTF-32	UTF-16
Solaris (SPARC)	32	UTF-32	UTF-16
Solaris (SPARC)	64	UTF-32	UTF-16
Solaris (x86)	32	UTF-32	UTF-16
Solaris (x86)	64	UTF-32	UTF-16

Solving Common Problems

This section contains information on debugging and troubleshooting your connector.

Process Can't Locate the DLL

A common cause of failure to connect to the data store is an ODBC connector or a server process that cannot find the ICU DLL or shared object and refuses to start. This can be frustrating to diagnose, so watch out for it. It is the general case of the connector not being able to find all of its dependencies.

On Windows, this manifests itself as a "-1 Error". One way to approach this problem is with the Dependency Walker program. This free program identifies the items on which an executable depends. For more information on dependency walker, see the MSDN article at http://msdn.microsoft.com/en-us/magazine/bb985842.aspx. The application can be installed from this location: http://dependencywalker.com/.

Connector Cannot Find the Data Store

Another cause of failure is the server or ODBC connector being unable to find your data store. This is usually a case of configuring the connector or server incorrectly. Make sure to include the right checks in your DSI implementation code to detect this condition, and to return clear error messages to the user. This is a frustrating problem to diagnose because the cause is often buried at the very bottom of the data access stack.

Incomplete Types Compiler Warning

In order to prevent the possibility of memory leaks when using class templates such as `AutoPtr`, `AutoArrayPtr` or `AutoValueMap`, a compiler warning will be generated when they are instantiated on an incomplete type. If you encounter a compiler warning about an incomplete type (the actual warning varies between compilers), simply include the header file of the pre-declared class, and remove the pre-declaration. This allows the compiler to have full access to the underlying class destructor of in the class template AutoXXX destructor.

Example: Code That Causes an Incomplete Type Warning

```
namespace MyDriver
{
    class MyClass1; // Pre-declaration of MyClass1
    class MyClass2
    {
        public:
        MyClass2 {}
        ~MyClass2 {}
        private:
        // when this is cleaned up, the destructor of MyClass1 will not be called :-(
        Simba::Support::AutoPtr<MyClass1> m_obj;
    }
}
```

Example: Resolving an Incomplete Type Warning

```
//To resolve the issue, include the header file for
// MyClass1 and remove the forward declaration:
#include MyClass1.h
namespace MyDriver
{
	class MyClass2
	{
		public:
		MyClass2 {}
		~MyClass2 {}
		private:
		// when this is cleaned up, the destructor of MyClass1 will be called :-)
		Simba::Support::AutoPtr<MyClass1> m_obj;
	}
}
```

Background on the Use of Incomplete Types

In C++, it is possible to pre-declare a class, then define pointer or reference to it. This results in a pointer to an incomplete type. This is fine as long as the code does not need to access any methods or attributes of the pre-defined class, including the destructor. The C++ specification also allows you to delete the pointer to an incomplete type. This may cause a problem, because the compiler does not know the type of the referenced object, or how to call its destructor (it might not even have a destructor). The compiler frees the memory of the object but cannot call its destructor first. This could lead to memory leaks or other issues, such as a file remaining open.

The Simba SDK provides class templates such as `AutoPtr`, `AutoArrayPtr` or `AutoValueMap` to help manage your dynamically created objects. These classes will clean up an object when their instances are destroyed. However, if these class templates are instantiated on an incomplete type, the compiler does not have access to the underlying class's destructor. Therefore, it cannot add a call to the destructor of the underlying object when compiling the destructor of these class templates.s

Incorrect version of libc Library

When deploying a connector on AIX platforms, a supported version of the system library `libc` must be available on the machine. We recommend having the following version of this library for each supported version of AIX:

AIX version	
AIX 7.1	bos.rte.libc.7.2.0.2

AIX version	
AIX 6.1	bos.rte.libc.6.1.9.30

To download this library, see http://www-01.ibm.com/support/docview.wss?uid=isg1fileset-870201775.

If this library does not exist on the deployment machine, the following errors may be encountered:

- [unixODBC][Driver Manager]Can't initiate unicode conversion
- [unixODBC][Driver Manager]Can't open lib *<path to connector library>*: file not found
- [ISQL]ERROR: Could not SQLConnect

Error Messages Encountered During Development

The following table lists some of the error messages you may encounter during the development and testing phases of your DSII. For a complete list of error codes and messages, see the files in `[INSTALL_DIRECTORY]\DataAccessComponents\ErrorMessages\`.

Error Message	Meaning	Solution
The license file <*file*> could not be found.	The `Simba.lic` license file is not in the correct directory, or you do not have a valid license.	Install the license file, or re-install it in the correct location. For information on installing the license file, see .
SQLDriverConnect returned: SQL_ ERROR=-1	The Driver Manager cannot find or load the requested connector's DLL.	Make sure that your connector is installed correctly and that the DSN is correctly configured.
Specified driver could not be loaded.	The connector is missing some dependencies. Another possibility is that all libraries have not been compiled with the same bitness. Check that your ICU, iODBC, and your DSII libraries are all the same bitness.	Try listing the dynamic dependencies of your connector and ensuring all the dependencies are available. e.g.: ldd -d driver.so on Unix or use Dependency Walker on Windows.
Your evaluation period has expired. Please contact Simba Technologies Inc. at support@simba.com	Your license has expired.	Contact Simba for support.

Error Message	Meaning	Solution
Error file not found: <file>	The error message file is missing or the configuration value used to locate the file was not set.	Ensure the ErrorMessagesPath configuration value exists and is pointing at the correct directory containing the error message files. On Windows, this configuration is in the registry at HKLM/Software/Simba/Driver and on other platforms it is found in the simba.ini config file.
The error message <message> could not be found in the en-US locale.	Same as above.	
Incomplete Type	A template class, such as `AutoPtr`, `AutoArrayPtr` or `AutoValueMap`, is used as a reference to an incomplete type.	See .

Contact Us

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

Note:

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

You can contact Technical Support at www.insightsoftware.com.

Third-Party Trademarks

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

All other trademarks are trademarks of their respective owners.

Third Party Licenses

The licenses for the third-party libraries that are included in this product are listed below.

ICU License - ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995-2014 International Business Machines Corporation and others

All rights reserved.

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.

All trademarks and registered trademarks mentioned herein are the property of their respective owners.

OpenSSL

Copyright (c) 1998-2011 The OpenSSL Project. All rights reserved.

1. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
2. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
3. 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.
4. All advertising materials mentioning features or use of this software must display the following acknowledgment:
5. "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
6. 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.

7. 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.

8. 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.

This product includes cryptographic software written by Eric Young(eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Original SSLeay License

Copyright (C) 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 adheared 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.

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

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.

2. 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.

3. All advertising materials mentioning features or use of this software must display the following acknowledge:

4. "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)"

5. The word 'cryptographic' can be left out if the rouines from the library being used are not cryptographic related :-).

6. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgment:
   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND ANY EXPRESS 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 AUTHOR OR 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.

The license and distribution terms for any publicly available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution license [including the GNU Public License.]

Expat License

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd

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 NOINFRINGEMENT. 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.

Stringencoders License

Copyright 2005, 2006, 2007

Nick Galbreath -- nickg [at] modp [dot] com

All rights reserved.

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.

Neither the name of the modp.com nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS 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 COPYRIGHT OWNER OR 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.

This is the standard "new" BSD license:

http://www.opensource.org/licenses/bsd-license.php

dtoa License

The author of this software is David M. Gay.

Copyright (c) 1991, 2000, 2001 by Lucent Technologies.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

CityHash License

CityHash, by Geoff Pike and Jyrki Alakuijala

Copyright (c) 2011 Google, Inc.

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.

http:code.google.com/p/cityhash/