



SimbaEngine SDK 9.0

Developer Guide

2012-01-31

Simba Technologies Inc.



Copyright © Simba Technologies Inc. All Rights Reserved.

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this publication, or the software it describes, may be reproduced, transmitted, transcribed, stored in a retrieval system, decompiled, disassembled, reverse-engineered, or translated into any language in any form by any means for any purpose without the express written permission of Simba Technologies Inc.

Simba Trademarks

Simba, the Simba logo, SimbaEngine, SimbaEngine C/S, SimbaClient, SimbaD20, SimbaEngine SDK and SimbaODBC are registered trademarks of Simba Technologies Inc. All other trademarks and/or servicemarks are the property of their respective owners.

Simba Technologies Inc.

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

Tel. +1.604.633.0008
Fax. +1.604.633.0004

www.simba.com

Printed in Canada

Third Party Trademarks

ICU License – ICU 1.8.1 and later

COPYRIGHT AND PERMISSION NOTICE

Copyright (c) 1995–2010 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) 1998–2008 The OpenSSL Project. All rights reserved.

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 above 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 acknowledgment:
"This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (<http://www.openssl.org/>)"
4. 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.
5. 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.
6. 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.

Expat

"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."

Table of Contents

1	Before you begin	8
1.1	Who should read this manual.....	8
1.2	Supported platforms.....	8
1.3	Conventions used in this manual.....	8
1.4	Abbreviations used in this manual.....	9
1.5	For more information	9
1.6	Contact us	10
2	Introducing SimbaEngine SDK.....	12
3	SimbaEngine Solutions	13
3.1	A Simple ODBC Driver	13
3.2	An ODBC Driver with a SQLEngine	14
3.3	Mix and match the components	15
3.4	How do the Simba components work together?	17
3.4.1	The Data Store Interface (DSI)	17
3.4.2	The Client/Server Protocol.....	17
3.5	How do I use the SDK?	19
3.5.1	Database Schema Translation	20
3.5.2	Preparing your workspace	21
3.5.3	Creating a database access solution.....	22
3.6	What's next?.....	22
3.6.1	Prototype data retrieval from your database.....	22
3.6.2	Learn more about the SDK.....	22
4	Using SimbaEngine SDK	23
4.1	Connecting components into a complete solution	23
4.2	Library Components.....	25
4.2.1	SimbaODBC.....	25
4.2.2	SimbaJDBC.....	25
4.2.3	Simba.NET.....	26
4.2.4	Simba SQLEngine	26
4.2.5	Client/Server.....	29
4.2.6	SimbaServerD20.....	32
4.2.7	C++ Bridges	32
4.3	Sample Drivers.....	32
4.3.1	Quickstart	32
4.3.2	DotNetQuickstart.....	33
4.3.3	JavaQuickstart	34
4.3.4	Codebase	34
4.3.5	Ultralight	35
4.3.6	DotNetUltralight	35
4.3.7	JavaUltralight.....	35
4.4	Using the Sample Drivers	35
4.4.1	Troubleshooting	36
5	Building Blocks for a DSI Implementation.....	38
5.1	Understanding the Building Blocks.....	41
5.1.1	Application Interface.....	42
5.1.2	Data Store Type.....	43
5.1.3	Local or Remote	43

5.1.4	Development Environments	43
5.2	Application Programming Interfaces.....	45
5.2.1	DSI API	45
5.2.2	DSI API Extensions.....	46
5.2.3	Lifecycle of DSI Objects	47
5.3	More depth on some of the basic features.....	49
5.3.1	Connection Parameters.....	50
5.3.2	Data Retrieval	50
5.3.3	Fetching Metadata	50
5.3.4	Add Custom Metadata Columns to your Metadata Results	55
5.4	Advanced Features.....	56
5.4.1	Collaborative Query Execution.....	56
5.4.2	Stored Procedures.....	61
5.4.3	DML	62
5.4.4	Transactions.....	62
5.4.5	ODBC Custom Data Types	64
5.4.6	JDBC Time and Timestamp with Timezone.....	68
5.4.7	JDBC Updatable ResultSets	68
5.5	Add Custom Connection & Statement Attributes.....	69
5.5.1	Java DSI Custom Properties	70
5.5.2	DotNet DSI Custom Properties.....	71
5.6	Overriding Default Property and Attribute Values.....	71
5.6.1	JDBC Specific Properties.....	73
5.6.2	ADO.NET Specific Properties	76
5.6.3	SQLEngine Specific Properties	76
6	Compiling your DSII	78
6.1	C++ under Windows.....	78
6.1.1	Build as an ODBC Driver (a DLL) for local connections.....	78
6.1.2	Build as a SimbaServer (an EXE) for remote connections	78
6.1.3	Building with Simba SQLEngine	79
6.1.4	Building without Simba SQLEngine	79
6.1.5	Run-time library options	80
6.1.6	Character Set.....	81
6.2	C# under Windows	81
6.2.1	DotNetDSI and DSII	81
6.2.2	Simba.NET.....	81
6.2.3	Build as an ODBC Driver (a DLL) for local connections.....	82
6.2.4	Build as a SimbaServer (an EXE) for remote connections	83
6.3	Java under Windows.....	83
6.3.1	Build as a JDBC Driver.....	83
6.3.2	Build as an ODBC Driver (a DLL) for local connections.....	84
6.3.3	Build as a SimbaServer (an EXE) for remote connections	84
6.3.4	Building with Simba SQLEngine	84
6.3.5	Building without Simba SQLEngine	85
6.4	Data Source Names and Driver entries in the Registry.....	85
6.5	C++ under Linux/Unix/MacOSX	86
6.5.1	Sample Makefile Options	86
6.5.1.1	Build as an ODBC Driver (a Shared Object) for local connections.....	87
6.5.2	Build as a SimbaServer (an Executable) for remote connections.....	87
6.5.3	Building with Simba SQLEngine	88

- 6.5.4 Building without Simba SQLEngine 88
- 6.5.5 Build configurations..... 89
- 6.6 Java under Linux/Unix/MacOSX..... 89
 - 6.6.1 Build as a JDBC Driver..... 90
 - 6.6.2 Build as an ODBC Driver (a Shared Object) for local connections..... 90
 - 6.6.3 Build as a SimbaServer (an Executable) for remote connections..... 91
 - 6.6.4 Building with Simba SQLEngine 91
 - 6.6.5 Building without Simba SQLEngine 91
- 6.7 Access to Data Sources under Linux/Unix/MacOSX..... 91
 - 6.7.1 Linux/Unix/MacOSX Driver Managers 91
 - 6.7.2 Configuring Data Sources under Linux/Unix/MacOSX 91
- 7 Testing your DSN 95
 - 7.1 Testing under Windows..... 95
 - 7.1.1 Microsoft Access..... 95
 - 7.1.2 Microsoft Excel..... 95
 - 7.1.3 ODBCTest 96
 - 7.2 Testing under Linux/Unix/MacOSX..... 97
 - 7.2.1 iODBCTest and iODBCTestW..... 97
 - 7.2.2 UnixODBC 98
 - 7.2.3 Logging 98
- 8 Packaging Your Driver..... 101
 - 8.1 Preparing your driver to ship to end users..... 101
 - 8.1.1 C++ Packaging for Windows..... 101
 - 8.1.2 C++ Packaging for Linux/Unix/MacOSX 102
 - 8.1.3 C# Packaging (Windows only) 103
 - 8.1.4 Java Packaging for Windows 104
 - 8.1.5 Java Packaging for Linux/Unix/MacOSX..... 105
 - 8.2 Create a Custom DSN Configuration Application..... 106
 - 8.3 How To Build Your Driver As An OEM Solution..... 107
 - 8.3.1 ODBC 107
 - 8.3.2 JDBC..... 108
 - 8.3.3 ADO.NET 109
- 9 Frequently Asked Questions..... 110
 - 9.1 What is ODBC? 110
 - 9.2 What is MDAC? 110
 - 9.3 I am new to ODBC. How does my application work with an ODBC Driver?..... 110
 - 9.4 What is ICU? 111
 - 9.5 What is SimbaODBC?..... 111
 - 9.6 What do the different components of SimbaODBC do? 112
 - 9.7 How do I build an ODBC driver using SimbaEngine? 112
 - 9.8 How can I obtain more information about SimbaEngine SDK?..... 112
 - 9.9 What SQL conformance level does SimbaEngine SDK support? 112
- Appendix A: Platforms and System Requirements 115
- Appendix B: Errors and Exceptions 116
- Appendix C: Using Critical Section Locks..... 119
- Appendix D: The Connection Process..... 120
- Appendix E: Creating and Using Dialogs 123

Appendix F: Posting Warnings.....	125
Appendix G: Data Types	126
Appendix H: Common Error Messages	130
Appendix I: Build Platforms and compilers.....	131
Appendix J: Driver Manager encodings on Linux/Unix/MacOSX.....	132
Appendix K: Sample Unix Makefiles to build as a Shared Object.....	133
Appendix L: Building a driver for UnixODBC 2.2.12 and earlier.....	143
Appendix M: Sample Unix Makefiles to build as a SimbaServer.....	144
Appendix N: Localization.....	149
Appendix O: Supported ODBC scalar functions.....	153

Figures and Tables

No table of figures entries found.

1 Before you begin

This guide contains detailed information on the SimbaEngine SDK, which gives you the tools you need to create a custom ODBC 3.52, JDBC 3.0 or ADO.NET driver. Your customers will use this driver to access the data in your system with common reporting applications.

1.1 Who should read this manual

This guide assumes you have a general understanding of ODBC architecture and ODBC driver components, and have access to the Microsoft ODBC Software Development Kit.

This document also assumes you have a working understanding of modern database principles and terminology; the C++, Java and/or C# languages; and your development environment.

1.2 Supported platforms

SimbaEngine SDK is available on a wide range of platforms including Windows, Linux, Solaris, HP-UX, AIX, and Mac OS X in both 32-bit and 64-bit versions. See Appendix A: Platforms and System Requirements on page 115 for details.

Using SimbaClient and SimbaServer, you can use a standards-based data driver to access your data on any platform, from any platform, regardless of processor architecture or word length. Whether your data is strictly ANSI or you have customers all around the World, SimbaEngine SDK can handle it with Unicode capability designed-in. Your new standards-based data driver will be ready to localize anywhere.

1.3 Conventions used in this manual

The full path of the installation directory for the SDK may vary depending on your system. In this document, we will represent it as [INSTALL_DIRECTORY], which defaults as follows (note that “9.0” is the current release and this part of the path will change with each release of the SDK):

- Windows platforms:
C:\Simba Technologies\SimbaEngineSDK\9.0
- Linux/Unix/MacOSX platforms:
<theUntarDirectory>/SimbaEngineSDK/9.0

Directories, files, and parameter names appear in italics. For example:

The libraries containing the data access components you need are in the *C:\Simba Technologies\SimbaEngineSDK\9.0\DataAccessComponents* folder.

Computer input and output, such as sample listings, messages that appear on your screen, and commands or statements that you are instructed to type, appear in Courier typeface. For example:

```
SQLDriverConnect returned: SQL_ERROR=-1.
```

Program and document names appear in narrow bold type when described in text. For example:

The document **SimbaEngine SDK Developer Guide**

Function names, SQL keywords, and source code appears in Courier typeface. For example:

```
// Create a log file if a connection has been established and
// a log file has not yet been created
if (m_connected)
{
    if (NULL == m_log)
    {
        m_log = new CustomerDSIILog(<name of the log file>);
    }

    return m_log;
}
```

1.4 Abbreviations used in this manual

Abbreviation	Full Text
APD	Application Parameter Descriptor
DSI	Data Store Interface
DSII	Data Store Interface Implementation
DSN	Database Source Name
ICU	International Components for Unicode
IPD	Implementation Parameter Descriptor

1.5 For more information

The following documentation is available for the SimbaEngine SDK:

- **SimbaEngine SDK Developer Guide** (*this document*): Detailed information on how to work with the SDK to develop an ODBC/JDBC/ADO.NET driver for virtually any data store.

- **Build a C++ ODBC Driver in 5 Days:** Condensed information to walk you through the process of creating a custom ODBC driver with the SimbaEngine SDK, using C++ as your development environment.
- **Build a C# ODBC Driver in 5 Days:** Condensed information to walk you through the process of creating a custom ODBC driver with the SimbaEngine SDK, using C# as your development environment.
- **Build a Java ODBC Driver in 5 Days:** Condensed information to walk you through the process of creating a custom ODBC driver with the SimbaEngine SDK, using Java as your development environment.
- **Build a JDBC Driver in 5 Days:** Condensed information to walk you through the process of creating a custom Type 4 JDBC driver with the SimbaEngine SDK, using Java as your development environment.
- **Build an ADO.NET Provider in 5 Days:** Condensed information to walk you through the process of creating a custom ADO.NET Data Provider with the SimbaEngine SDK, using C# as your development environment.
- **SimbaEngine DSI API Reference Guide:** Detailed information about function parameters, return types and error and message codes.
- **SimbaClientServer Users Guide:** Detailed information explaining the creation, installation, configuration, and administration of the client and server components of the SimbaEngine SDK.
- **SimbaEngine SDK Release Notes:** Information about incremental changes introduced in a particular release of the SimbaEngine SDK.

For complete information on the ODBC 3.5 specification, see the MSDN ODBC Programmer's Reference, available from the Microsoft web site at: [http://msdn.microsoft.com/en-us/library/ms714562\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms714562(VS.85).aspx)

For complete information on the ADO.NET classes, see Accessing Data with ADO.NET, available from the Microsoft web site at: [http://msdn.microsoft.com/en-us/library/e80y5yhx\(v=VS.71\).aspx](http://msdn.microsoft.com/en-us/library/e80y5yhx(v=VS.71).aspx)

For complete information on the JDBC 3.0 specification, see the JDBC(TM) API Specification, available from the Sun Developer Network web site at: <http://www.oracle.com/technetwork/java/javase/jdbc/index.html>

1.6 Contact us

We welcome your questions and comments and we want you to succeed. To help us help you faster please have ready a detailed summary of your machine environment (operating system, version, patch-level, etc.) before you contact us. Providing us with this information helps us to understand your situation and to help you more effectively.

On the Web:

Visit us on the Web at www.simba.com.

Submit technical requests online at www.simba.com/support.htm
or send e-mail to support@simba.com

By telephone:

Simba support is available from 9:00 AM to 5:00 PM Pacific Time, Monday to Friday.
Dial +1.604.633.0008 and select 3.

By fax:

Fax: +1.604.633.0004

2 Introducing SimbaEngine SDK

SimbaEngine SDK enables you to create data access solutions for any data store – including those that can process SQL and those that do not understand SQL at all, such as object-oriented, ISAM and SCADA data sources. In fact, if you can make your data store look like it uses tables and columns, SimbaEngine SDK can create a driver for it that your customers can use with common reporting applications.

SimbaEngine SDK is a collection of database access tools packaged into a small set of components. The components fit together in different ways to solve a variety of data access problems. Included in SimbaEngine SDK are components based on widely used and supported data access standards like ODBC, JDBC, ADO.NET and SQL. This means that the data access solutions you create can be used by common reporting applications like Crystal Reports and Microsoft Excel, as well as a wide variety of enterprise, specialized and custom applications.

Collaborative Query Execution

With the unique Collaborative Query Execution technology in SimbaEngine SDK, you can have your execution engine take over execution of any part of a query. For remote data stores this speeds up query processing by reducing the amount of data that Simba SQLEngine must retrieve. For high-performance data stores, like column-oriented databases or key-value stores, this allows you to expose the high-performance of your data engine while Simba SQLEngine only executes the remaining portions of the query. This unique capability allows Simba SQLEngine and your high-performance data store to work collaboratively to deliver the highest performance to your customers.

Data Access Standards

You can build remote and local ODBC 3.52, JDBC 3.0 and ADO.NET drivers using a single Data Store Interface (DSI) implementation to connect SimbaEngine SDK to your data store. This lowers your development and testing costs and allows you to focus on delivering value to your customers. A simple, iterative, development approach allows you to quickly deliver standards-based data access to your customers, and later add performance and functional improvements, as they demand them. You can strategically assign your development resources to features that your customers want.

3 SimbaEngine Solutions

Solutions created with SimbaEngine SDK have a clear, simple structure that makes them easy to understand. This section introduces the structure of the solutions you create with SimbaEngine SDK and explains how the different parts work together. As you continue to learn about SimbaEngine SDK, you can relate your newfound knowledge to your understanding of the standard structure. Note that in the diagrams of SimbaEngine SDK solutions below, program control flows downward and retrieved data flows upward.

3.1 A Simple ODBC Driver

Building a driver for a SQL-capable data store

Figure 1, below, is a diagram of an ODBC driver created with SimbaEngine SDK to connect to an SQL data store. In this case, the Customer Data Store understands SQL-92, or a variant. The application creates SQL queries and sends them to the ODBC driver, where they are possibly modified and sent on to the data store. The data store executes the SQL queries and creates a result set. The ODBC driver can then move the result set from the data store back to the application.

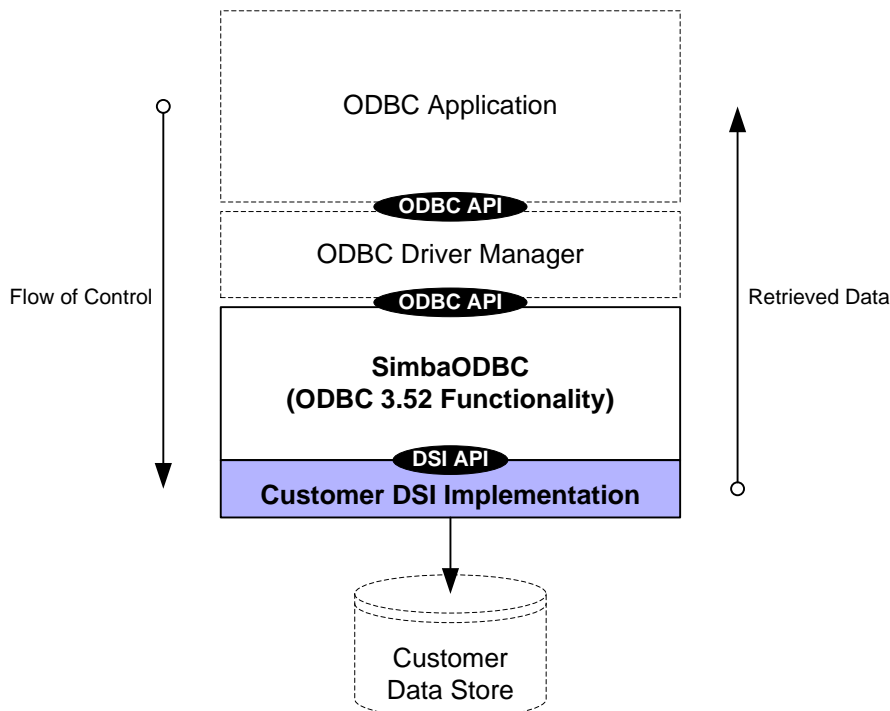


Figure 1: The architecture of an ODBC driver built with SimbaODBC.

There are several things to note in Figure 1. First is the layered, component nature of the system, with well-defined interfaces between the components. The ODBC API is the standard

interface used by most data access applications, such as Microsoft Excel and Crystal Reports, to access relational data stores. The ODBC API hides the idiosyncrasies of the data store from the application so the application can be written once and be able to connect to many different data stores. The ODBC Driver Manager is a shared library (.DLL, .SO) that mediates between the application and the ODBC driver. It performs error checking and translation that widens the range of the ODBC drivers that can be used by the application. Microsoft supplies the Driver Manager on Windows. Simba can supply a driver manager for non-Windows operating systems.

In Figure 1, the two components separated by the Simba Data Store Interface (DSI) API make up the ODBC driver itself. The majority of the functionality is contained in the SimbaODBC component. This component implements all the functionality of ODBC so that you don't have to. This includes error checking, the driver, session and statement management, and keeping track of the ODBC details expected by the Driver Manager and the application. Simba supports and maintains the SimbaODBC component as part of SimbaEngine SDK and makes sure that you don't have to worry about any changes to the ODBC API or the way applications use it.

The SimbaODBC component communicates with the "Customer DSI implementation" via the Data Store Interface, or DSI. This interface is common to all SimbaEngine SDK components that communicate with customer code. The customer's DSI implementation (DSII) is the component that connects directly to the data store and it is specific to each different data store. It is custom designed by you for each different data store and its interface.

3.2 An ODBC Driver with a SQLEngine

Building a driver for a non-SQL data store

Many data stores do not understand SQL. In fact, many data stores are not organized as tables and columns at all. For example, object-oriented, network and SCADA data stores have non-tabular storage, and a database system that allows users to query familiar business objects, such as invoices and shipments, may not store its data as those entities at all. In these cases, you must add a SQLEngine to the system to enable ODBC applications to access the data store. Figure 2, below, is a diagram of this kind of ODBC driver.

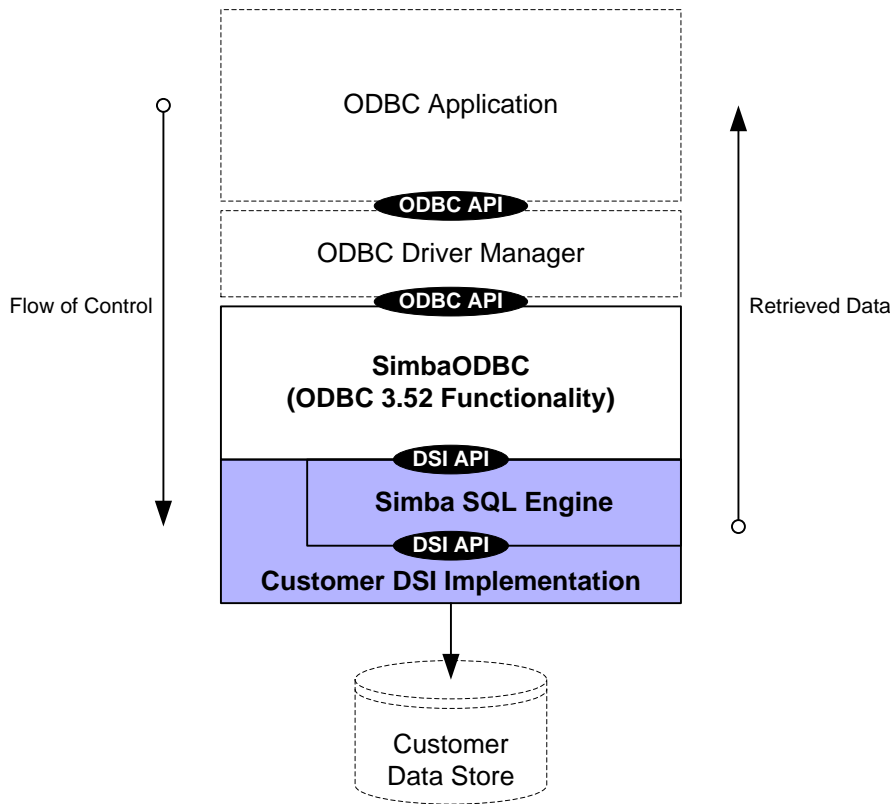


Figure 2: The architecture of an ODBC driver built with SimbaEngine.

In Figure 2, you can see the familiar SimbaODBC component and the DSI API, but there is a new component inserted between them – Simba SQL Engine – that provides the SQL-92 processing required for ODBC and other standard interfaces. The DSI API remains the same and performs the same service in this case as it did in Figure 1, but you can now use common reporting applications to access non-SQL data stores as well.

The key idea is that SimbaEngine SDK provides access to both SQL and non-SQL data stores using its simple layered architecture. This architecture expands to cover remote data access (client/server) whether supplied by SimbaEngine SDK or the customer data store, many industry standard data access interfaces, and data stores available through managed languages such as Java and C#. The architecture remains the same and the DSI API provides the data store abstraction to allow many different solutions with one DSI implementation.

3.3 Mix and match the components

One DSI implementation yields many solutions

Figure 3, below, shows three different solutions from SimbaEngine SDK. Note that even though the issues solved are very different, the architecture of the solutions essentially remains the same. Example A is the simplest, with SimbaODBC and the Customer DSI implementation

combining to provide ODBC access to a local, SQL-capable data store. Example B shows the architecture from Figure 2 but this time the Customer DSI implementation is combined with the Simba SQL Engine and access a remote data store. Example C shows the Simba Client/Server components in use to deliver standard data access to remote applications. Again, the same basic architecture applies and the same Customer DSI implementation created for Example B can be used in Example C.

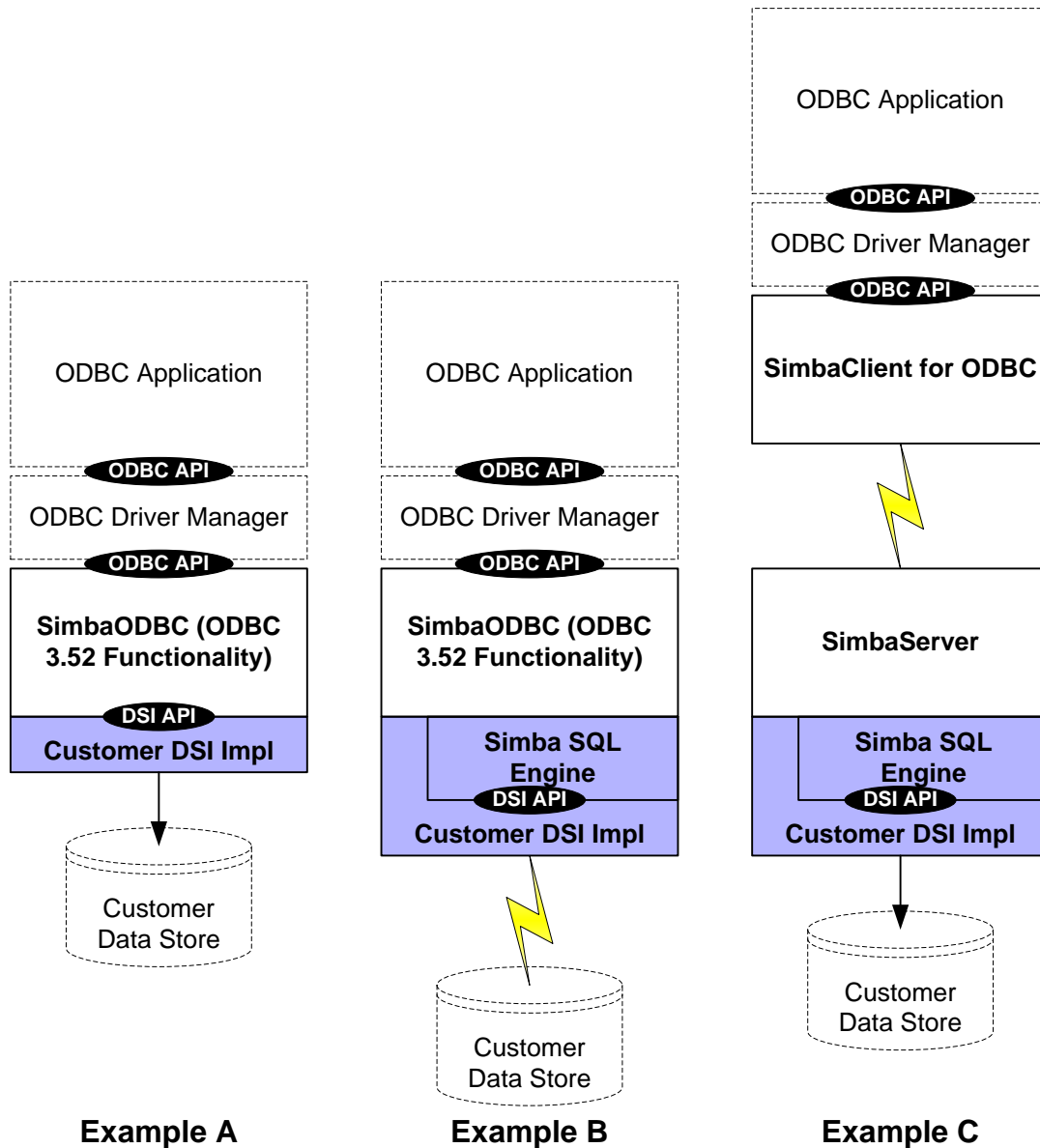


Figure 3: Three examples of SimbaEngine SDK architecture for comparison.

3.4 How do the Simba components work together?

How do they communicate?

The real secret of the components in SimbaEngine SDK is that they can all connect together using two simple interfaces. We have designed the interfaces so they do exactly what is required, which allows them to be simple and efficient. Using only a few interfaces means more flexibility in the way they can connect together. The following subsections describe this.

3.4.1 The Data Store Interface (DSI)

A canonical database access interface

The Data Store Interface defines a generic view of an SQL database that is independent of data access standards such as ODBC, JDBC and ADO.NET. SimbaEngine SDK translates the ODBC, JDBC, and ADO.NET interfaces to the DSI in C++, Java, or C#. The DSI API is object-oriented and simpler to use than the industry interfaces so it is easier to write a DSI implementation that will translate to a custom data store. Depending on your data store, you can use a single implementation of the DSI in several ways, such as those illustrated by Figure 3 above, in which each of the different architectures shown has the DSI API and a DSI implementation at the bottom of its stack. The DSI API also provides a consistent API whether you are implementing ODBC, JDBC, or ADO.NET, and thus makes it easier to re-use your knowledge when creating a driver for a different interface.

The DSI serves as a common API and SimbaEngine SDK includes code to map from standard data access interfaces to the DSI. If you write code to map from the DSI to your data store, you are creating a driver from one of the standard application interfaces. SimbaEngine SDK can also make the other interfaces available via its Client/Server component, and so with a single DSI implementation you can make your data store accessible to the other standard application interfaces as well.

In Chapter 5, “Building Blocks for a DSI Implementation” on page 38, you will learn more about the concept of the DSI and how all of the components work together. For detailed information about the DSI API method calls, please see the SimbaEngine DSI API Reference Guide found in the Documentation folder in the installed SimbaEngine SDK.

3.4.2 The Client/Server Protocol

Another simple interface used by SimbaEngine SDK is the 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. Figure 4, below, shows the architecture of the stack created with the Simba ODBC Client, Simba Server and a DSI implementation. Simba Client communicates with Simba Server using the Simba Client/Server protocol. Simba Server

translates the Simba Client/Server protocol to the DSI API, and any DSI implementation can be linked to the Simba Server to provide remote data access.

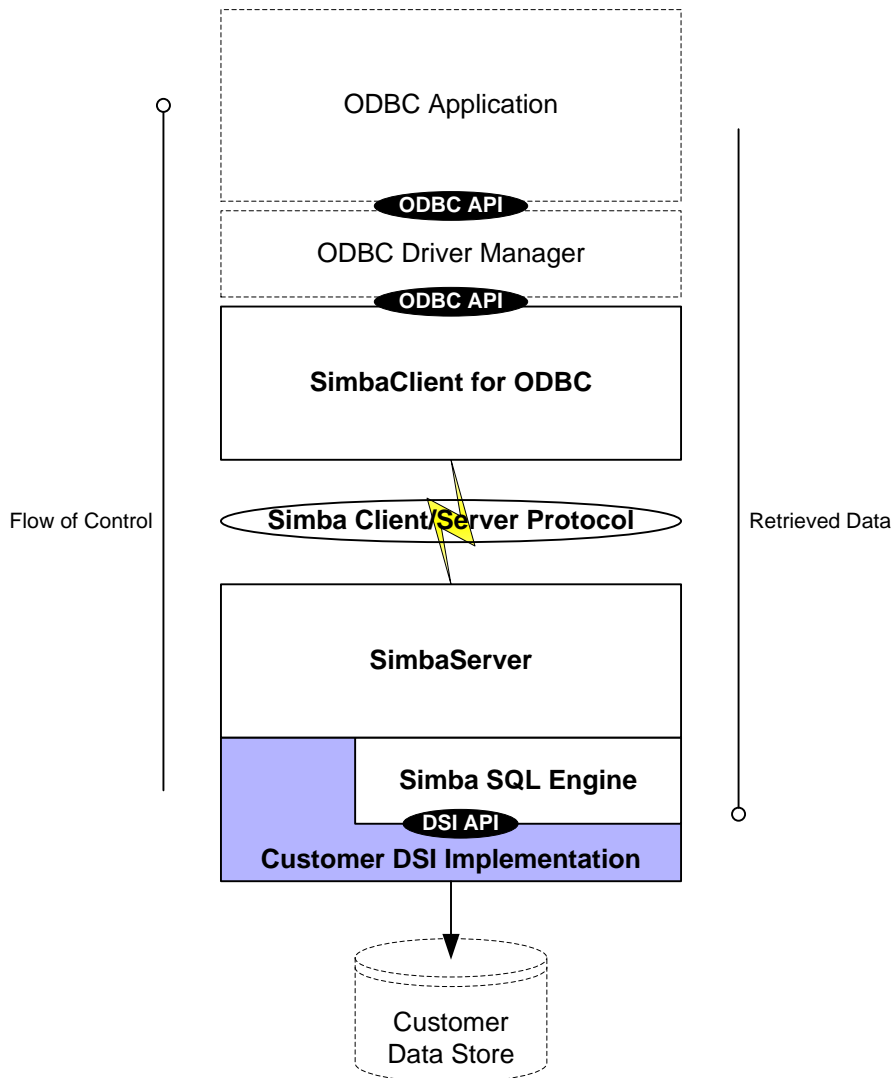


Figure 4: The Simba Client/Server architecture showing the SimbaClient for ODBC connecting remotely to Simba Server.

As an aside, SimbaClient uses the DSI API internally. SimbaClient is the combination of SimbaODBC linked to a DSI implementation that translates the DSI API to the Simba Client/Server protocol.

Figure 5, below, shows the architecture of three different stacks that each start with a different type of application: ODBC, ADO.NET and Java. Each client translates its specific data access interface to the Simba Client/Server protocol for transmission to the server.

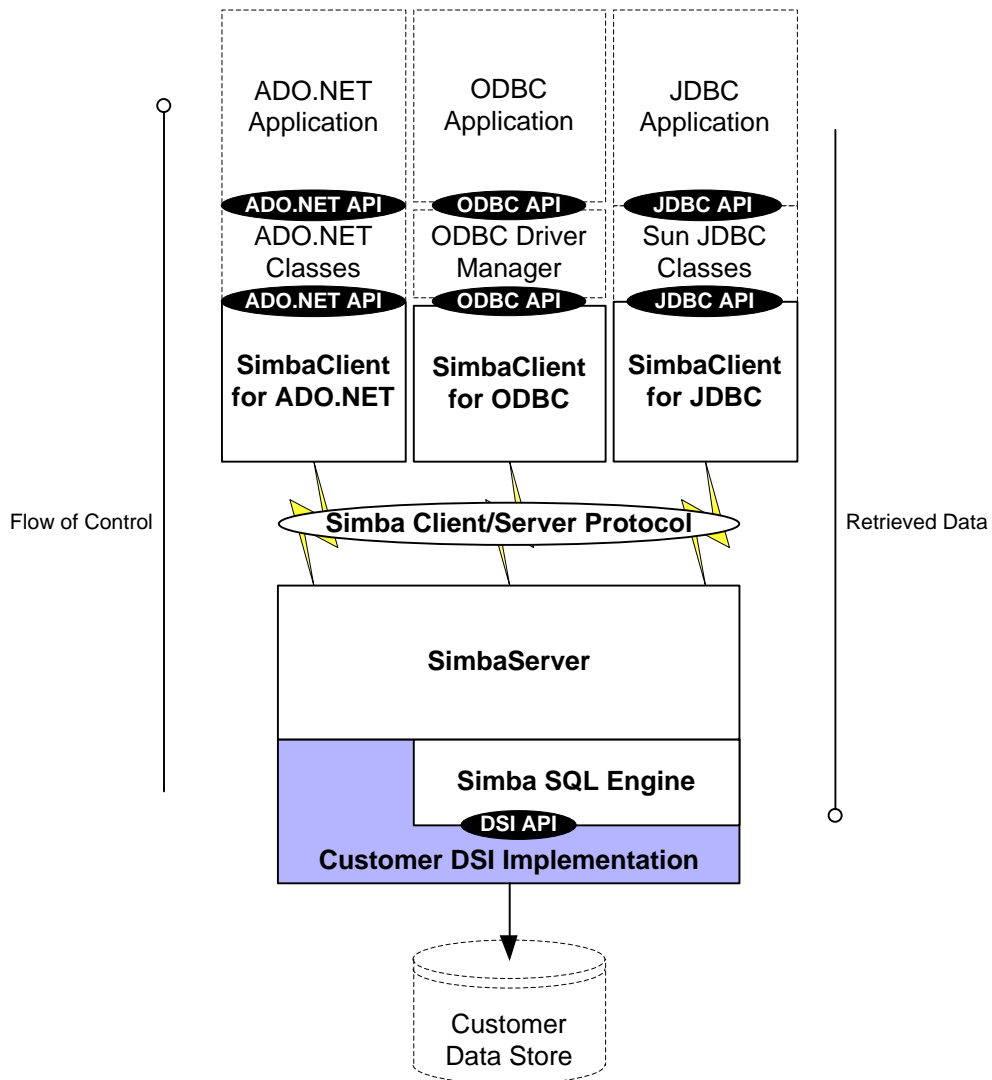


Figure 5: A comparison of three different data access standard clients connecting remotely to Simba Server.

3.5 How do I use the SDK?

Whether you are at the proof-of-concept stage and taking the route to build a sample driver in just a few days, or you are ready to build a complete commercial driver for your data source that you will ship to your customers, you will follow the same general steps:

1. Plan how to translate your data store schema to the DSI view of the world.
2. Make a copy of the appropriate driver folders (`SimbaEngineQuickstart`, `SimbaEngineCodebase` or `SimbaEngineUltralight`) and rename them.
3. Implement your plan for translating your data store to the DSI.

Of course, your planning and implementing activities will take longer for a commercial driver than for a proof-of-concept. However, the difference is largely the amount of detail required. In both cases, the goal is to arrive at a working executable that is either a driver or a server that you can test with a common reporting tool.

3.5.1 Database Schema Translation

How do I make my data store work with SimbaEngine SDK?

The first step is to plan how you will translate your data store schema to the DSI view of the world. The DSI represents the data store as a series of tables and columns, and the purpose of your DSI implementation is to translate the real data store schema into the DSI representation. If you can imagine this translation then you can make SimbaEngine SDK work for you.

Figure 6, below, illustrates visually the kind of translation required. Consider an existing database that uses an object-oriented or networked schema to store the data because it suits the primary purpose of the database. 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 SimbaEngine can execute SQL queries against it. In this way, any database you can represent as tables and columns can be accessed by SimbaEngine and made accessible to common reporting tools.

Translating a Network Database into Tables and Rows

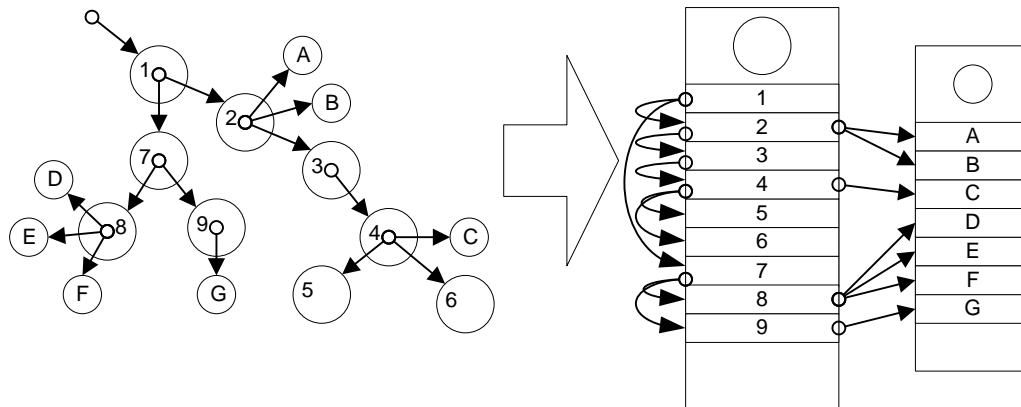


Figure 6: An imaginary non-tabular database schema is represented by tables and columns containing the same data and linked in the same way. The two databases are equivalent, but relational applications through Simba SQLEngine can directly address the tabular database.

One simple approach you can use to create a tabular view of your data store is to read the entire database into temporary tables. You can then access the temporary tables through the DSI. However, this is inefficient and realistically only works for very small databases. A more efficient method is to create virtual tables and then access the original database when SimbaEngine requests data through the DSI. There are many approaches and the one you choose must suit your data store.

3.5.2 Preparing your workspace

When you know how to translate your data store schema to the DSI view of the world, the next step is to create a workspace for yourself where you can begin work with your new DSI implementation. To work on a new DSI for SimbaEngine, select the Simba-provided sample that you want to start with, copy the entire folder and rename it to something appropriate for you (this is described in detail for each of the different development platforms in the Build a Driver in 5 Days document series). The `SimbaEngineCodebase` and `SimbaEngineQuickstart` samples both use the Simba SQLEngine, while the `SimbaEngineUltralight` sample does not.

If, for example, you choose the “`SimbaEngineQuickstart`” example your starting point will be the source code and the project files or make files to build a complete, working SimbaEngine Quickstart sample DSI implementation that will read text files. Your job is to transform this source code so your data store supplies the data instead of the sample’s text files.

Also included is source code for an ODBC configuration DLL you can modify. This configuration DLL allows your customers to create and modify ODBC DSNs that use your new ODBC driver to connect to your data store.

3.5.3 Creating a database access solution

What does my driver look like when it is done?

The goal is to compile and link your DSI implementation into a DLL or shared object that common reporting applications can use to access your database. In the process, the project files or make files will link in the appropriate SimbaODBC and SimbaEngine libraries to complete the driver. If you are building a client/server solution, the project files or make files will also link in the appropriate Simba Client/Server libraries to complete the server executable. In all cases, the goal is to create an executable file that can be accessed by common reporting applications and will access your data store when SimbaEngine executes an SQL statement.

In the final executable, the components from SimbaEngine SDK take responsibility for meeting the data access standards while your custom DSI implementation takes responsibility for accessing your data store and translating it to the DSI API.

3.6 What's next?

Now that you have a high-level understanding of the SimbaEngine solutions, you have three options for what to do next:

1. Start prototyping right away.
2. Learn more about SimbaEngine SDK.
3. Dig deeper into the architecture and theory behind SimbaEngine.

3.6.1 Prototype data retrieval from your database

If you are ready to start exploring SimbaEngine SDK and making a first try to access your data store with SimbaEngine SDK, open up one of the three versions of the Build a Driver in 5 Days documents (there is one for each DSII language you might chose: C++, Java and C#). These provide step-by-step instructions to create a working driver using the SimbaEngine Quickstart sample driver. It is an excellent way to learn about SimbaEngine SDK while exploring what you need to do to connect it to your data store.

3.6.2 Learn more about the SDK

If you have already prototyped a driver for your data store, or if you want to learn more to help you plan a full commercial driver, continue reading Section 4, Using SimbaEngine SDK, below. This section goes into more detail about the internal workings of SimbaEngine SDK and its components, such as SimbaODBC and Simba SQLEngine. These components perform complicated work to make it easy for you to access your data store, but there are still things you need to consider when building a commercial driver. Things like data types, caching and pass-down optimizations are all easier to understand and take advantage of if you learn some of the theory and architecture of SimbaEngine SDK.

4 Using SimbaEngine SDK

Writing code, building drivers

We have designed SimbaEngine SDK to help you create data access solutions that make it possible for common reporting tools, like Crystal Reports and Microsoft Excel, to query your data store. If your data store does not understand SQL then the solution will include Simba SQLEngine to process SQL queries. If your data store does understand SQL then Simba SQLEngine should not be included and SimbaEngine SDK tools will pass on the SQL queries. SimbaEngine SDK provides the components to create a front end to your data store that includes a complete implementation of ODBC 3.52, JDBC 3.0 or ADO.NET. In addition, there are also SimbaClient and SimbaServer components that allow you to create a solution that accesses your data store remotely.

All of these components are part of the core libraries installed with the SDK. A series of project files are also included—these are sample drivers, which you can copy and modify to create your own first DSI Implementation.

This section describes the components and how they can be connected together to create a complete data access solution. It also describes the different sample drivers and outlines how you work with them.

4.1 Connecting components into a complete solution

Components + Interfaces = Executables

The goal of SimbaEngine SDK is to create an executable file that will run and enable access to your data store. This can be a Windows DLL, a Linux or Unix shared object, a stand-alone server, or some other form of executable. You create the executable by linking libraries from SimbaEngine SDK with a DSI implementation that you have written for your data store. What kind of executable you create depends on the solution you need, but you can use your DSI implementation in different ways. You can create several different solutions, but you only need to create one DSI implementation for your data store and it can be employed in each.

SimbaEngine SDK components and your DSI implementation link together through internal interfaces. Together they present external, standard interfaces for common applications to use. The DSI API and the Client/Server protocol are internal interfaces used to link components together into a single solution. The DSI API links your DSI implementation to SimbaEngine SDK components. This is usually a static link but it can be a dynamic link if required. The Client/Server protocol is an internal interface connected at run time when a SimbaClient wants to connect to a SimbaServer. All SimbaClients connect and communicate with SimbaServer using the same Client/Server protocol.

SimbaEngine SDK uses third-party components to perform common functions. Two are:

- SimbaEngine SDK uses International Components for Unicode (ICU) to convert between ANSI and Unicode representations of text, and between the various Unicode encodings. SimbaEngine SDK components call ICU functionality using ICU's own interface. The component is dynamically loaded on Windows, Linux and Unix.
- OpenSSL is an implementation of the Secure Sockets Layer and the Transport Layer Security protocols as well as a full-strength cryptography library. SimbaEngine SDK uses OpenSSL to secure the network connection between SimbaClient and SimbaServer.

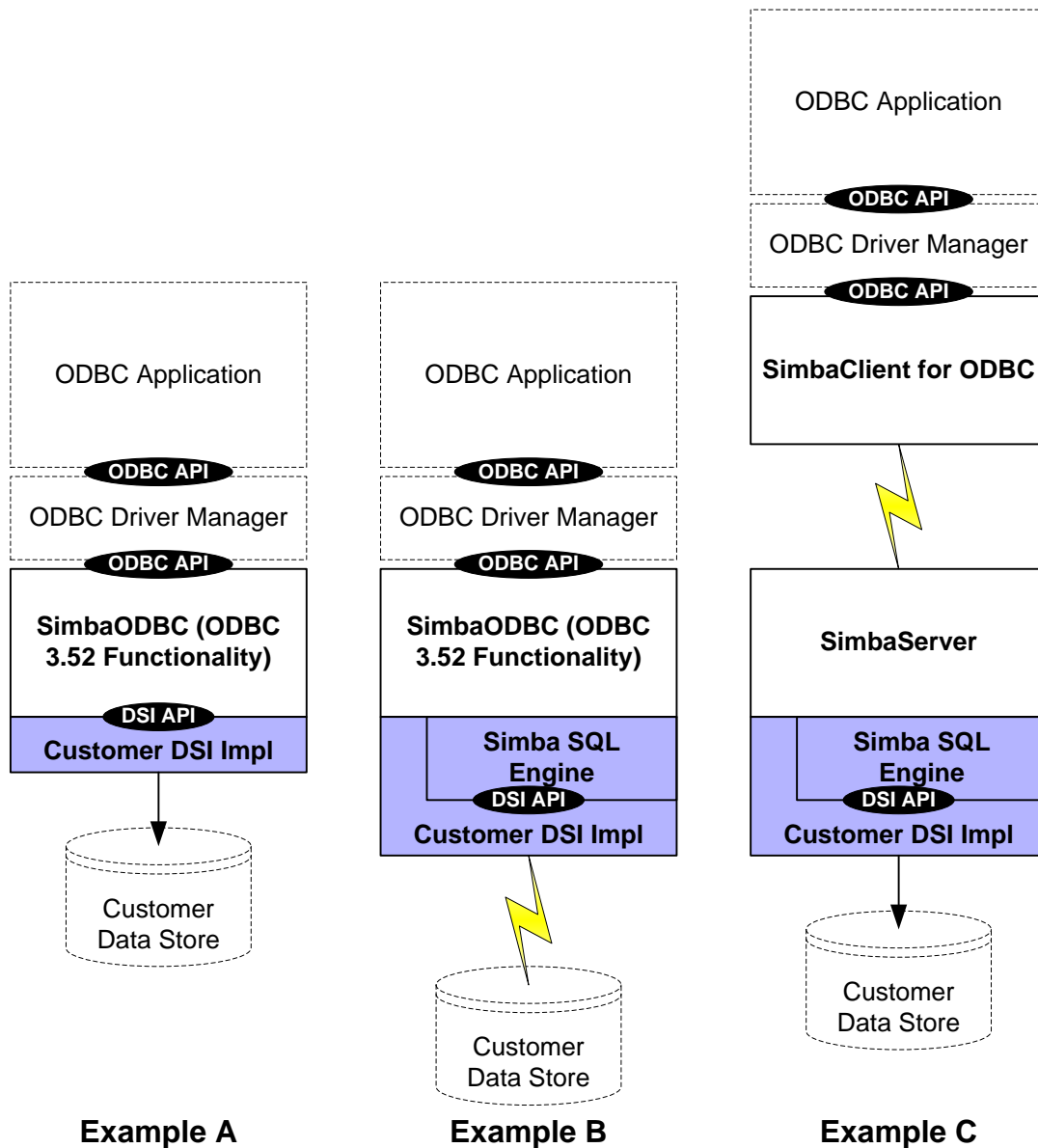


Figure 7: Three examples of system stacks you can build with SimbaEngine SDK.

Figure 7 illustrates three possible ways you can combine SimbaEngine SDK components to create different solutions for different situations. The completed data access solution presents a standard external interface to applications, such as ODBC, JDBC and ADO.NET.

Connecting from applications, configuring a data source

Standards-based applications connect to these application interfaces at run time, and the connections are sensitive to proper configuration. Sometimes, as in the case of a JDBC or ADO.NET connection, the application itself handles the connection configuration via information handed to the standard interface at connect time. In the case of ODBC, a Data Source Name, or DSN, maintains the configuration information, and the application refers to the DSN at connect time. To change the connection configuration in the DSN, the user runs the Windows Data Source Administrator to execute a special configuration DLL included with each ODBC driver. With the configuration DLL, the user can change the configuration of a DSN within the specifications of the driver. Incorrectly configured DSNs are frustrating, so some ODBC configuration DLLs include a connection test so that you can check the configuration immediately.

4.2 Library Components

This section describes the individual components that are available to you in SimbaEngine SDK. Later on (in section 5, “Building Blocks For A DSI Implementation” on page 38), we will show you how they all fit together and help you determine which components you need for your unique solution.

4.2.1 SimbaODBC

SimbaODBC provides a complete ODBC 3.52 interface and all the processing required to meet the ODBC 3.52 specification. We designed it as the connection between your custom DSI implementation and common reporting applications such as Crystal Reports and MS Excel. All that is required to create a complete ODBC driver is a DSI implementation that connects to an SQL database. SimbaClient for ODBC is an example of a driver like this. You can create a more complicated driver for non-SQL data by adding Simba SQLEngine to the driver. The SimbaEngine Codebase Driver is a provided sample of this kind of driver.

For detailed information about the DSI API method calls, please see the SimbaEngine DSI API Reference Guide found in the Documentation folder in the installed SimbaEngine SDK.

4.2.2 SimbaJDBC

SimbaJDBC provides a complete JDBC 3.0 interface and all the processing required to meet the JDBC 3.0 specification. We designed it as the connection between your custom DSI implementation and common JDBC reporting applications. All that is required to create a complete JDBC driver is a Java DSI implementation that connects to an SQL database.

SimbaClient for JDBC is an example of a driver like this. You can create a more complicated driver for non-SQL data by using SimbaClient for JDBC along with SimbaServer and the JNI DSI API.

4.2.3 Simba.NET

Simba.NET provides a complete ADO.NET interface and all the processing required to meet the ADO.NET specification. We designed it as the connection between your custom DSI implementation and common ADO.NET reporting applications such as Microsoft Analysis Services. All that is required to create a complete ADO.NET provider is a DotNet DSI implementation that connects to an SQL database. SimbaClient for ADO.NET is an example of a driver like this. You can create a more complicated driver for non-SQL data by using SimbaClient for ADO.NET along with SimbaServer and the CLI DSI API.

4.2.4 Simba SQLEngine

Simba SQLEngine is a self-contained SQL parser and execution engine. It consumes ODBC SQL-92 queries, parses them, creates an optimized execution plan, allows your DSI implementation to take over part or all of the execution, and then executes the plan against the DSI implementation. We have uniquely designed Simba SQLEngine to be sandwiched between two instances of the DSI rather than exposing another API. The top end of SQLEngine is compatible with the SimbaODBC or SimbaServer DSI specification and you can link it directly to either of them to create a stand-alone ODBC driver or a server. The bottom end of Simba SQLEngine is compatible with another part of the DSI specification. It calls into a DSI implementation that connects to a non-SQL data store. The result is SQL processing added to a non-SQL data store and made available to common reporting tools through standard interfaces.

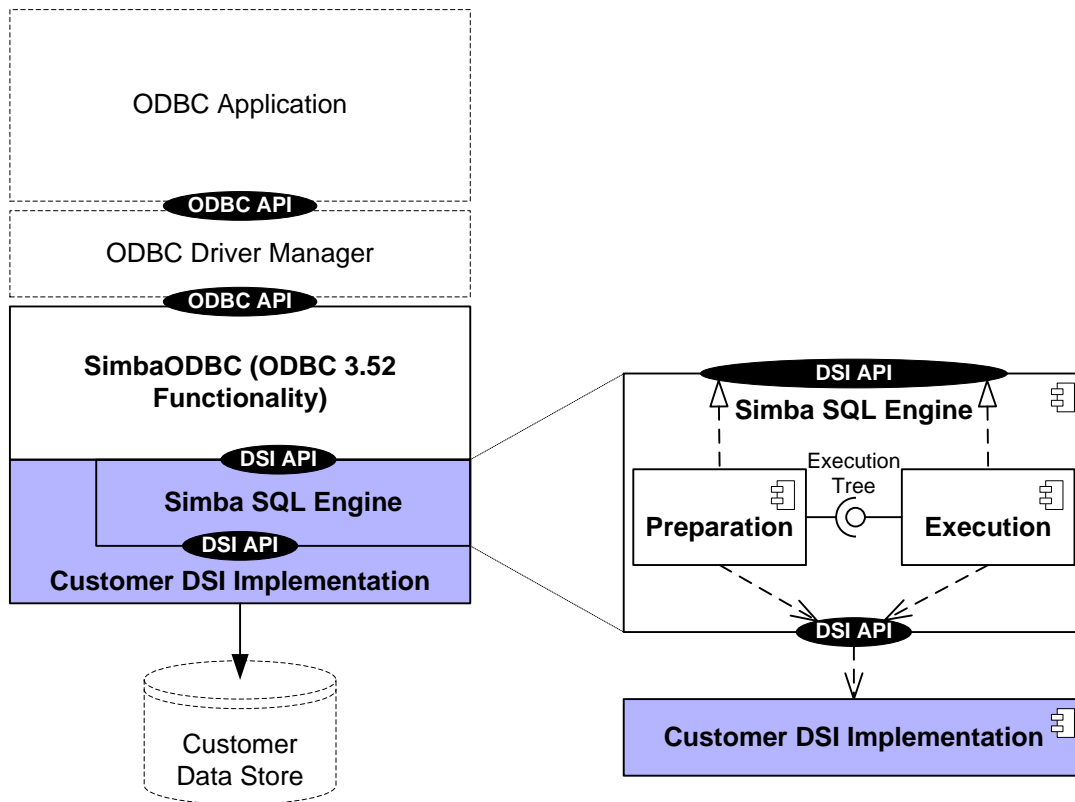


Figure 8: A look inside Simba SQL Engine showing the Preparation and Execution components communicating via the Execution Tree.

Figure 8, above, illustrates the architecture of Simba SQL Engine at a high-level. The Preparation component contains the SQL parser. It also validates the SQL by making sure that data store objects such as tables and columns exist. Once the SQL statement is prepared it is handed to the Execution component. The Execution component provides the environment for executing the execution tree and retrieving the result set.

Simba SQL Engine expects to have its non-SQL data store presented to it as tables and columns. This is part of the relational, SQL view of data, and the job of the DSI implementation is to translate from your actual data storage schema to a table and column view. Put another way, if you can translate your data store into a view with tables and columns, then the addition of Simba SQL Engine will create a relational RDBMS. If the data store will support it, an RDBMS created with Simba SQL Engine will support DDL, DML, and DCL SQL statements.

Note that Simba SQL Engine is only available when used with SimbaODBC. If a SQL enabled JDBC or ADO.NET driver is required, then using SimbaClient for JDBC or SimbaClient for ADO.NET along with SimbaServer can achieve this.

How can I help Simba SQLEngine to optimize queries?

If the data store implements indexes of some kind, or if it is able to find specific data rows very quickly, then Simba SQLEngine can take advantage of this data store functionality to optimize operations such as filtering and sorting. Note that the data store does not actually have to implement indexes in the traditional, ISAM, sense. As long as the data store can seek to specific data rows as if it was using an index (that is, faster than the SQLEngine could step through the data itself), then Simba SQLEngine can use that functionality as if it was a real index. Simba SQLEngine also uses table cardinality and other metadata to optimize the query before execution. If indexes and table cardinality are not available, Simba SQLEngine will still work but it will be slower because it will not be able to perform the more advanced optimizations.

Simba SQLEngine uses the index and metadata information in a sophisticated optimizer to find the lowest-cost execution plan. However, many data stores have high performance features that are part of the value they deliver to users. With Simba SQLEngine's Collaborative Query Execution, you can use these features to accelerate the execution of SQL statements under Simba SQLEngine.

What kind of processing will Simba SQLEngine let me do in my data store?

Before it executes an SQL statement, Simba SQLEngine can pass to the DSI implementation an optimized representation of the SQL statement, called an Algebraic Expression Tree, or AE-Tree. The SQL statement takes this form just before Simba SQLEngine transforms it into an execution plan and executes it. When Simba SQLEngine passes the AE-Tree to the DSI implementation, the DSI implementation can choose to execute any part of the AE-Tree itself. It signals its intentions by modifying the AE-Tree before returning it to Simba SQLEngine. For instance, if the data store can filter data, or join data, or execute aggregate functions particularly fast, it can modify those nodes of the AE-Tree to point to the DSI implementation for execution. The DSI implementation can modify any part of the AE-Tree if it can perform the execution quickly, or it can replace the entire tree and execute the whole query itself.

SELECT Col1 FROM Table1 WHERE Col2 = 23

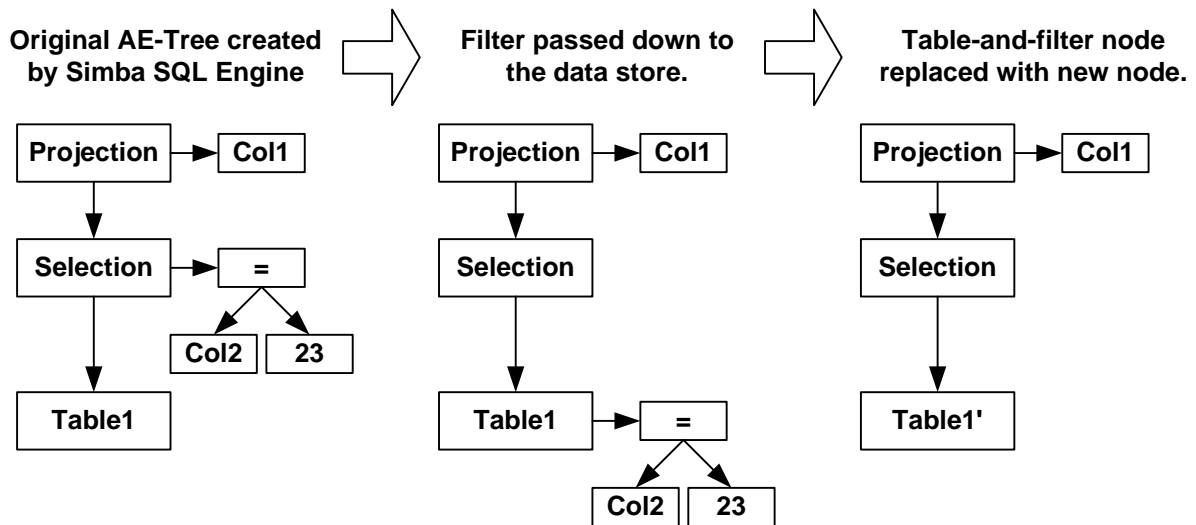


Figure 9: The progression of creating an AE-Tree, passing down a filter to the data store, and replacing the original table node with a new filtered table node.

This process is illustrated in Figure 9, above, which shows three views of a notional AE-Tree corresponding to the simple SQL query “SELECT Col1 FROM Table1 WHERE Col2 = 23”. The first view shows the AE-Tree originally created by Simba SQL Engine. In this case, all the columns in the projection are retrieved and filtered by the Simba SQL execution engine. In the second view, the filter has been passed down to the DSI implementation code and the filter function removed from the selection node. Now the DSI implementation is responsible for filtering the rows it returns to Simba SQL Engine. The DSI implementation replaces the “Table1” node with a new node, “Table1’ ”, that only returns the filtered row set.

After the DSI implementation passes back the AE-Tree, Simba SQL Engine transforms the modified AE-Tree into an execution plan and executes it. Simba SQL Engine execution engine, and the DSI implementation and data store collaborate on the execution of the SQL statement, with the data store executing the parts it can do quickly, and Simba SQL Engine executing the rest. Of course, the DSI implementation does not have to modify the AE-Tree at all. Simba SQL Engine can execute the entire SQL statement relatively quickly and efficiently by itself.

4.2.5 Client/Server

Simba Client/Server is a collection of smaller components that allow remote access to your data store. 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. You must link SimbaServer to a DSI implementation before it can be an executable. The DSI implementation can include Simba SQL Engine or not, and it can be written to perform a wide range of functionality including SQL query processing with Simba SQL Engine, 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.

Running SimbaClient and SimbaServer

When you start up your linked SimbaServer, it binds to a configurable port on your server machine and listens for connection requests from SimbaClient drivers. Since all SimbaClient drivers use the same protocol they are all handled in the same way by SimbaServer. When a SimbaClient driver finds SimbaServer, it requests a connection. With a successful connection, the SimbaClient and SimbaServer begin a conversation using the Simba Client/Server protocol. This is a layered protocol designed for clients making remote data queries and optimized for transmitting the result sets back to the client. It is independent of the standard interface used by the user application. Figure 10, below, shows the relationship of SimbaClient and SimbaServer using SimbaClient for ODBC as an example.

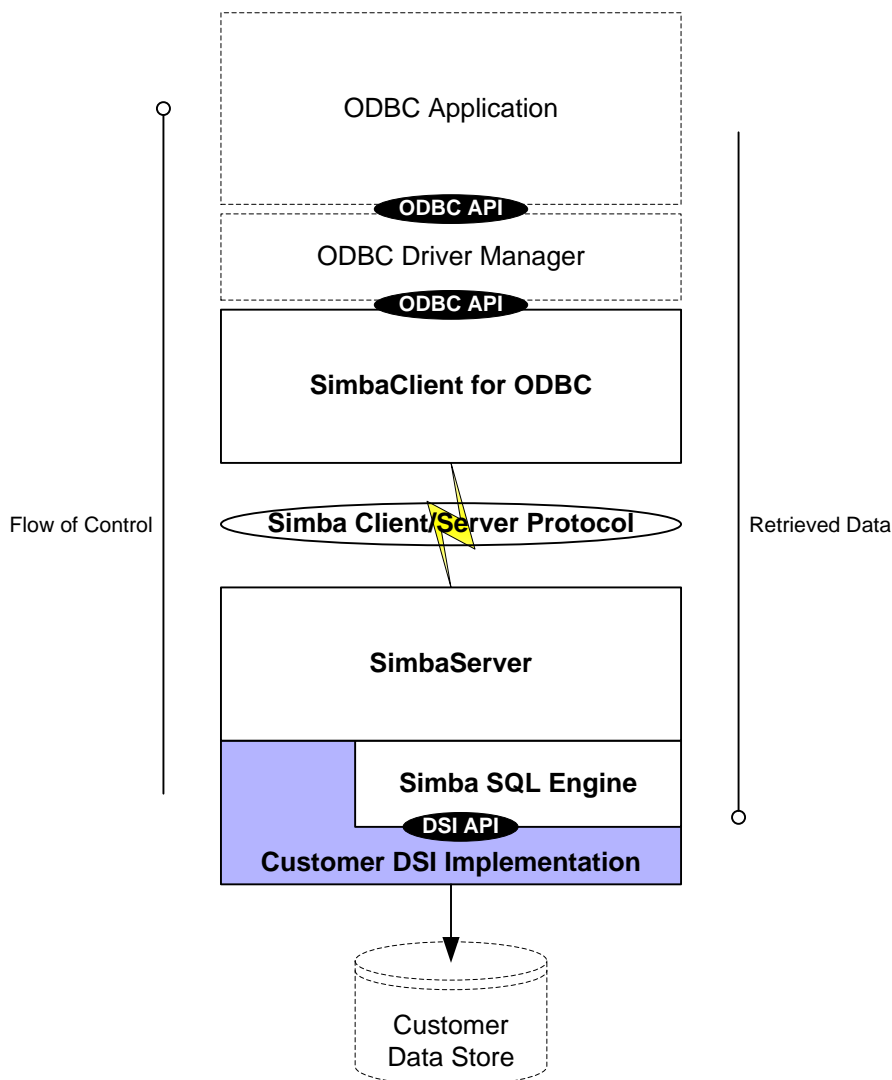


Figure 10: The architecture of Simba Client/Server showing the Client/Server protocol connecting the components.

SimbaServer is designed to optimize use of shared server resources, while SimbaClient is designed to optimize the responsiveness of the application to give the best experience to the user. The protocol parameters are configurable in case the default parameters do not provide the best performance for your circumstances.

Reusing your DSI Implementation

One of the important design features of SimbaServer is that it links downward to exactly the same DSI implementation as SimbaODBC. This means that you can first develop and test your DSI implementation as a local stand-alone ODBC driver using SimbaODBC. This is a simpler initial environment than developing using SimbaServer. When your new DSI implementation is performing to your satisfaction, you can link it to SimbaServer and begin testing it in a remote way. If you know the state of the logic and performance of the DSI implementation before introducing client/server, you can reduce your investigation time and debugging costs.

SimbaEngine SDK contains a SimbaClient for ODBC, a SimbaClient for JDBC, and a SimbaClient for ADO.NET that provide direct access to SimbaServer.

SimbaClient for ODBC

SimbaClient for ODBC is an ODBC driver 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 SQL Engine in the stack will be on the server side, there is no need for Simba SQL Engine in this driver. This is a completely generic ODBC driver that, when queried, reports the capabilities of the database that is connected to SimbaServer. There is nothing to modify in this ODBC driver.

SimbaClient for JDBC

SimbaClient for JDBC is a JDBC 3.0 driver 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. There is nothing to modify in this JDBC driver.

SimbaClient for ADO.NET

SimbaClient for ADO.NET is an ADO.NET driver DLL that can connect to SimbaServer. It requires the DotNet DSI and Simba.NET components to also be installed in an end user's environment. SimbaClient for ADO.NET includes the equivalent of SimbaODBC and custom ADO.NET code that communicates via the Simba Client/Server protocol with SimbaServer. There is nothing to modify in this ADO.NET driver.

4.2.6 SimbaServerD20

This is a stand-alone server that provides a bridge to any pre-existing ODBC enabled driver. It acts as an application on the server machine and talks to the DSN indicated by the ServerDSN connection key. You use it in conjunction with any SimbaClient, 64 or 32, C++ or JDBC, on any platform. There are no special actions needed to use SimbaD20 with a driver, aside from setting ServerDSN, starting the server, and connecting with a client.

4.2.7 C++ Bridges

C++ to Java Bridge (JNI DSI)

This component of the SDK allows you to use a Java development environment to write the data access portion of your driver and then link upwards to SimbaODBC or SimbaServer (including Simba SQLEngine) to create an ODBC driver.

C++ to C# Bridge (CLI DSI)

This component of the SDK allows you to use a C# development environment to write the data access portion of your driver and then link upwards to SimbaODBC or SimbaServer (including Simba SQLEngine) to create an ODBC driver.

4.3 Sample Drivers

4.3.1 Quickstart

Quickstart is a C++ sample DSI implementation of an ODBC driver 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 driver that you can copy and transform into a driver that accesses your non-SQL data store. An ODBC configuration DLL is included.

Also included is a version that has been compiled as a stand-alone server executable. For this version, the build system has linked Simba SQLEngine upward against SimbaServer instead of against SimbaODBC. The server accepts connection requests from SimbaClients and supports the Simba Client/Server protocol. You can use this build for remote testing of your Simba SQLEngine DSI implementation after you have got it working as an ODBC DLL or shared library.

We designed the Quickstart driver for prototyping DSI implementations so you can quickly understand the way SimbaEngine SDK works and get hands-on experience. However, you can also use it as the foundation for your commercial DSI implementation if you are careful to remove the shortcuts and simplifications that it contains. This is a fast and effective way to get a data access solution to your customers.

Quickstart data files

For its data store, the Quickstart example uses tabbed Unicode text files that can be created by MS Excel. With Simba SQLEngine, it is fully capable of retrieving, filtering and joining data from these tables, but the tables limit the data types supported and the performance. The functionality of the Quickstart example allows you to make incremental changes to the code and then test to make sure that everything still works. This is a proven way to understand the working of the DSI implementation and to prototype a DSI implementation for your own data store. This is also the best way to explore what you need to translate the schema of your data store to the tables and columns representation required by the DSI API.

There is a well-developed series of steps to take to get a prototype DSI implementation working with your data store: setting up the development environment, making a connection to the data store, retrieving metadata, working with columns, retrieving data, and writing data. At the end of each step, you can verify that the code you have written is working by testing the driver against your data store. At the end of five days, you can have a read-only driver connecting to your data store. This will not be a commercial driver, but you will have learned a lot about SimbaEngine SDK, and if you want to you can move on to add advanced capability and optimizations to create a first version of your commercial driver. Read the document, “Build a C++ ODBC Driver in 5 Days” to follow this methodology.

Verifying suspected defects in SimbaEngine SDK

Another purpose of the Quickstart sample DSI implementation is to provide a canonical DSI implementation that can be used to analyze and debug problems encountered. For instance, if you think your DSI implementation is working correctly but there is a problem in your SimbaEngine 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 Quickstart sample DSI implementation, then the problem lies likely in SimbaEngine SDK components. In addition, we can easily duplicate and fix the problem. If the problem goes away when you replace your DSI implementation with the Quickstart sample, 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.

4.3.2 DotNetQuickstart

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). Read the document, “Build a C# ODBC Driver in 5 Days” for a step-by-step walk-through of the process of creating a custom ODBC driver using C# as your development environment.

4.3.3 JavaQuickstart

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 JNI DSI API). Read the document, "Build a Java ODBC Driver in 5 Days" for a step-by-step walk-through of the process of creating a custom ODBC driver using Java as your development environment.

4.3.4 Codebase

Codebase is a C++ sample DSI implementation of a more advanced ODBC driver that reads an ISAM-like database. As with Quickstart, this is not a SQL-aware sample data source, so the Simba SQL Engine component is used to perform the necessary SQL processing.

The Codebase driver is designed to provide examples of features that you can examine, copy and adapt to your own situation. Much of this functionality has to be in all DSI implementations so it is useful to see it in action. The kinds of functionality illustrated in the Codebase sample DSI implementation are:

- Driver-wide information
- Opening tables
- Retrieving data from tables
- Virtual tables
- Metadata – gathering it and retrieving it
- Table cardinality (for optimization)
- Data caching in the DSI implementation
- Collaborative Query Execution optimization of filters, joins and aggregation

The Codebase data store is compatible with dBase files and supports indexes and several data types. An ODBC configuration DLL is included.

Also included is a version that has been compiled as a stand-alone server executable. For this version, the build system has linked Simba SQL Engine against SimbaServer instead of SimbaODBC. The server accepts connection requests from SimbaClients and supports the Simba Client/Server protocol. You can use this build for remote testing of your Simba SQL Engine DSI implementation after you have it working as an ODBC DLL or shared library.

Note that the SimbaEngine Codebase sample is only available on Windows.

4.3.5 Ultralight

Ultralight is a sample driver 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.

4.3.6 DotNetUltralight

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 driver will be written entirely in C#, providing an ADO.NET interface. When using Simba's C++ to C# bridge API, the resulting driver will be a mixture of C# and C++, providing an ODBC interface or SimbaServer executable for use with any of the SimbaClient drivers.

4.3.7 JavaUltralight

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 driver will be written entirely in Java, providing a JDBC 3.0 interface. When using Simba's C++ to Java bridge API, the resulting driver will be a mixture of Java and C++, providing an ODBC interface or SimbaServer executable for use with any of the SimbaClient drivers.

4.4 Using the Sample Drivers

To test any of these examples by building and running them, the procedure is approximately the same—this is a generalized version of Day One from the Build a Driver in 5 Days documents (there is one for each supported language, C++, Java and C#):

1. Navigate to the appropriate Source folder. E.g.
[INSTALL_DIRECTORY]\Examples\SimbaEngineQuickstart\Source
2. Open the solution file (e.g. QuickstartDSII_net2008.vcproj) in Visual Studio if you are using C++ or C# on Windows; or edit the make file if you are using Java or if you are on a Linux/Unix environment.
3. Verify that the configuration is what you want – 32 or 64-bit, Debug or Release, and static or dynamically linked.

4. Build the executable.

If the executable you compiled is a stand-alone ODBC driver:

1. Use the installed DSN to connect to the driver.
2. Verify that you can see tables by retrieving metadata (`SQLTables`, `SQLColumns`).
3. Verify that you can see the data by executing a simple query like `SELECT * FROM T1 (SQLExecDirect)`.

If the executable you compiled is a SimbaServer:

1. Install one of the SimbaClients – ODBC can be the most convenient.
2. Start the server process.
3. Create a DSN on the client machine (can be the same machine as the server machine).
4. Confirm that the client can connect to the server.
5. Verify that you can see tables by retrieving metadata (`SQLTables`, `SQLColumns`).
6. Verify that you can see the data by executing a simple query like `SELECT * FROM T1 (SQLExecDirect)`.

To build your own driver using the SimbaEngine Quickstart driver as a starting point, begin by making a copy of the appropriate folder and renaming it. Then follow the “Build a Driver in Five Days” instructions, modifying your newly created copy of the SimbaEngine Quickstart sample. You can continue to add and test new functionality to your SimbaEngine Quickstart sample. The most commonly used functions are the use of insert and update, delete, create and drop tables. To build a commercial driver you might want to pursue a more formal path with analysis and design before doing too much implementation, because the early decisions you make can effect later decisions and implementation.

Unit testing, developer testing

One of the most effective tools for testing an ODBC driver is ODBCTest, a GUI application that allows the developer to make any ODBC call, with any parameter value, in any order. The results of each call are displayed – errors or retrieved data and metadata. See section 7.1.3, “ODBCTest” on page 96 for more information. There are also similar programs that work on Linux/Unix platforms. See Section 7.2, “Testing under Linux/Unix/MacOSX” for more information.

4.4.1 Troubleshooting

A common cause of failure to connect to the data store is an ODBC driver 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 driver not being able to find all its

dependencies. On Windows, this manifests itself as a -1 Error. One way to approach this problem is with Dependency Walker. This free program identifies the items on which an executable depends. There is an article in MSDN about it here <http://msdn.microsoft.com/en-us/magazine/bb985842.aspx> and you can find the application here <http://dependencywalker.com/>.

Another cause of failure is the server or ODBC driver being unable to find your data store. This is usually a case of configuring the driver 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.

5 Building Blocks for a DSI Implementation

SimbaEngine Components

The important diagrams in this section illustrate the components available in the SDK, how they fit with each other and how they fit with different application and data store technologies. There are two variations, Figure 11 illustrates the components when your data store is local (or you have your own remoting protocol), and Figure 12 changes the view to the perspective of a remote data store, using Simba’s ClientServer components and remoting protocol.

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 or JDBC driver, or ADO.NET provider that you build with the SDK.
SimbaEngine SDK	These elements, shown in white boxes, are the components that make up the SDK itself. For a description of each of the components shown here, please see section 4.2, “Library Components”.
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, as shown in Figure 12: Component Diagram – Remote Data Store, 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, as shown in Figure 12: Component Diagram – Remote Data Store, 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, as shown in Figure 12: Component Diagram – Remote Data Store, 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.

Local Data Store

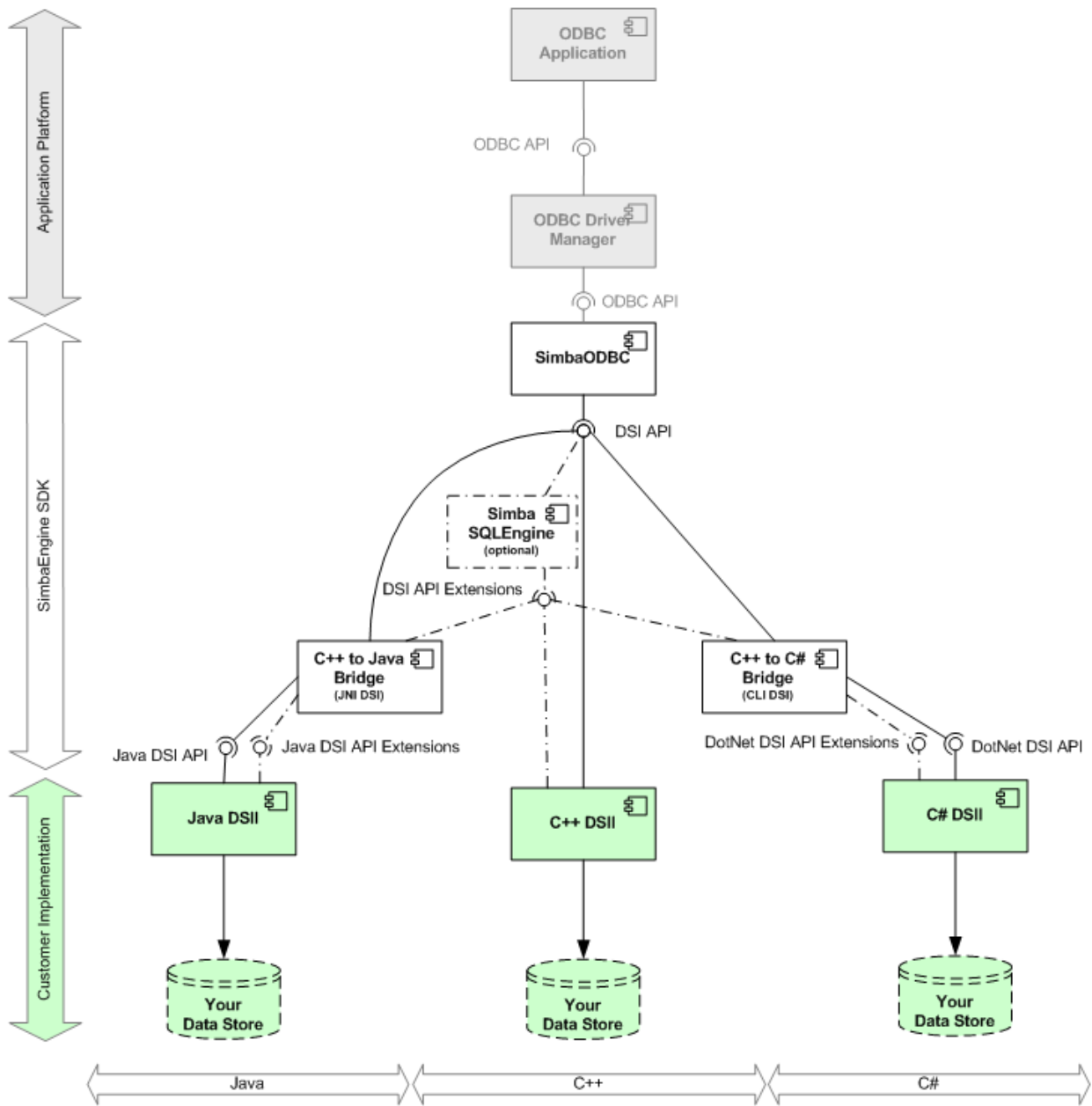


Figure 11: Component Diagram – Local Data Store

Remote Data Store

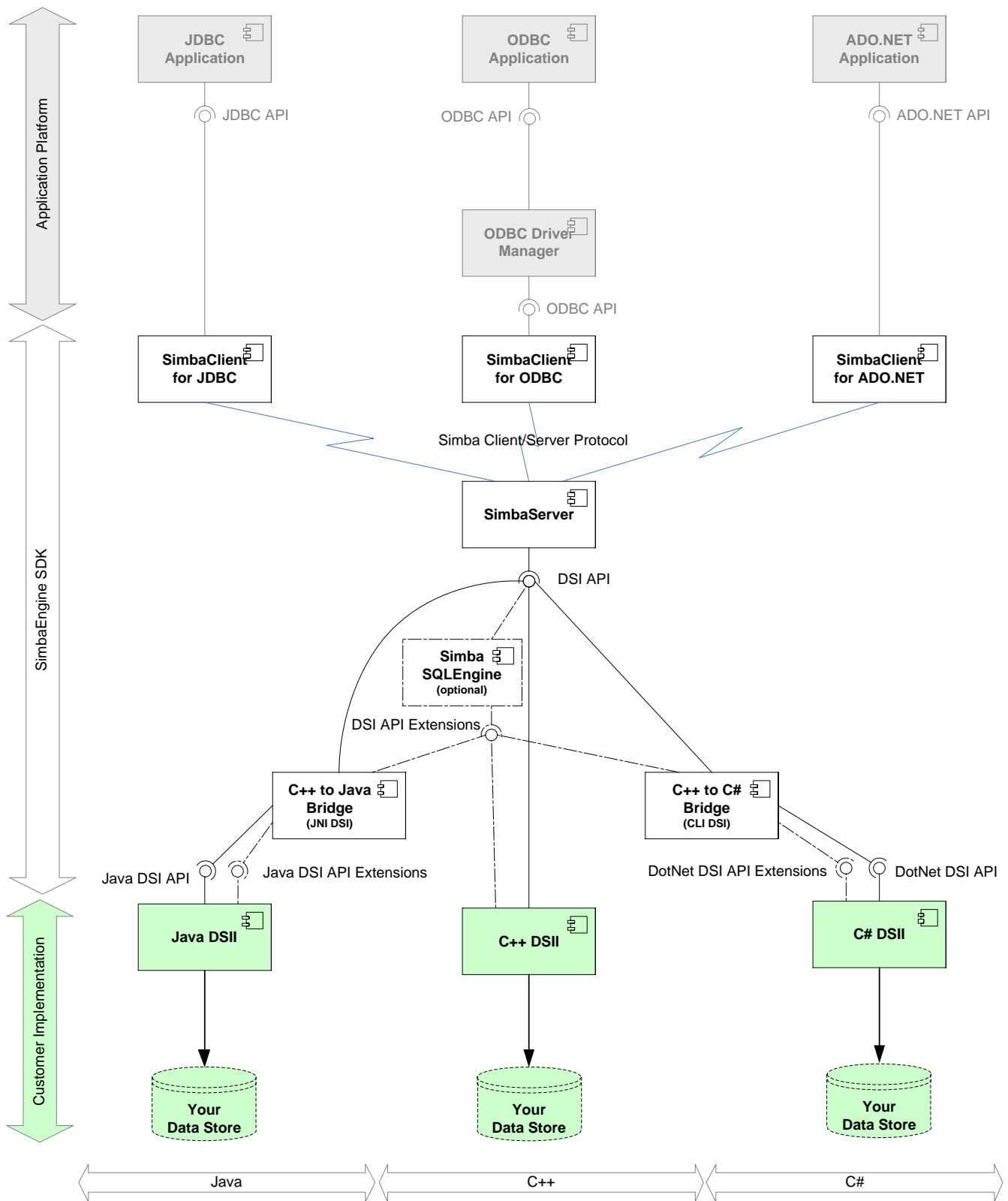


Figure 12: Component Diagram – Remote Data Store

5.1 Understanding the Building Blocks

You can narrow down all of the choices available to with SimbaEngine SDK by answering the following four questions:

1. Will your end users' applications access data sources via ODBC, JDBC or ADO.NET?
2. Is your data store SQL-capable or not?
3. Will you need local or remote access to your data store?
4. Will your driver DSI implementation environment be C++, Java or C#; on Windows or Linux/Unix?

Table 1, below, identifies all the possible combinations of these options and identifies the resulting API that you will use from the SDK in that particular case. It also identifies the sample drivers that demonstrate each combination.

Application Interface	Dev. Env't	Data Store Type	Sample Driver(s)	API
ODBC	C++	SQL, Local	Ultralight	DSI API
ODBC	C++	SQL, Remote	Ultralight + SimbaServer	DSI API
ODBC	C++	Non-SQL, Local	Quickstart, Codebase	DSI API + Extensions
ODBC	C++	Non-SQL, Remote	Quickstart + SimbaServer, Codebase + SimbaServer	DSI API + Extensions
ODBC	Java	SQL, Local	JavaUltraLight	Java DSI API + JNI DSI
ODBC	Java	SQL, Remote	JavaUltraLight + SimbaServer	Java DSI API + JNI DSI
ODBC	Java	Non-SQL, Local	JavaQuickstart	Java DSI API + JNI DSI + Extensions
ODBC	Java	Non-SQL, Remote	JavaQuickstart + SimbaServer	Java DSI API + JNI DSI + Extensions
ODBC	C#	SQL, Local	DotNetUltraLight	.NET DSI API + CLI DSI
ODBC	C#	SQL, Remote	DotNetUltraLight + SimbaServer	.NET DSI API + CLI DSI
ODBC	C#	Non-SQL, Local	DotNetQuickstart	.NET DSI API + CLI DSI + Extensions
ODBC	C#	Non-SQL, Remote	DotnetQuickstart + SimbaServer	.NET DSI API + CLI DSI + Extensions
JDBC	Java	SQL, Local	JavaUltraLight	Java DSI API
JDBC	Java	SQL, Remote	JavaUltraLight + SimbaServer	Java DSI API + JNI DSI
JDBC	Java	Non-SQL, Remote	JavaQuickstart + SimbaServer	Java DSI API + JNI DSI + Extensions
ADO.NET	C#	SQL, Local	DotNetUltraLight	.NET DSI API
ADO.NET	C#	SQL, Remote	DotNetUltraLight + SimbaServer	.NET DSI API + CLI DSI

Application Interface	Dev. Env't	Data Store Type	Sample Driver(s)	API
ADO.NET	C#	Non-SQL, Remote	DotNetQuickstart + SimbaServer	.NET DSI API + CLI DSI + Extensions

Figure 13: Table of Permutations and Combinations of SimbaEngine Components

5.1.1 Application Interface

When building a driver that will access your data store locally, your choice of application interface is closely tied to your development environment, as outlined below. Note, however, that using Simba Client/Server for remote access to your data store, as shown in the diagram above, the wire protocol between SimbaServer and SimbaClients detaches the client application from the driver that accesses the data store. Your DSI Implementation is completely independent of your client solution—any client type can talk to a SimbaServer, regardless of what language is used to access the underlying data store.

Developing for ODBC client applications

If you plan to build your local driver for client applications that will use ODBC, SimbaEngine SDK offers you three options for the development environment you will use to access your data store. C++ is the most common choice, but the SDK provides bridge API's to both C# and Java, allowing you to work in the best environment for your needs.

Developing for JDBC client applications

To support JDBC applications, your driver can (as shown in the diagram above) be written to use SimbaJDBC or SimbaClient for JDBC and SimbaServer. For local drivers, you may use either SimbaJDBC or SimbaClient for JDBC, and SimbaServer, on the same machine. When using SimbaJDBC, your driver must be written in Java, but when using SimbaServer your driver implementation can be written in Java, C++, or C#.

Developing for ADO.NET client applications

To support ADO.NET applications, your driver can (as shown in the diagram above) be written to use Simba.NET or SimbaClient for ADO.NET and SimbaServer. For local drivers, you may use either Simba.NET or SimbaClient for ADO.NET, and SimbaServer, on the same machine. When using Simba.NET, your driver must be written in C#, but when using SimbaServer your driver implementation can be written in Java, C++, or C#.

5.1.2 Data Store Type

When would your solution require the use of SQLEngine?

This subject is covered in some detail in section 3.1 (“A Simple ODBC Driver” on page 13) and in section 3.2 (“An ODBC Driver with a SQLEngine” on page 14), but to quickly re-state the main distinctions here:

- If you are planning to access a flat file, an object-oriented database, a SCADA-like process database, or an ISAM-like traditional database, you will link to the Simba SQLEngine libraries to provide the SQL processing needed by ODBC.
- If your database is already SQL-aware, your driver will pass the SQL directly through to your own data engine component.

5.1.3 Local or Remote

When would your solution follow one of the local models?

- Client applications will access a database that runs on each user’s own machine (perhaps your product is a client management database and you are giving your customers an ODBC driver so they can do more charting of their data).
- You have already configured your database for network access and some component of your software is already installed on user machines. Your new driver 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 driver 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.

When would your solution follow one of the remote models?

- Your software runs on a server and users access it via a web user interface. Your new driver using SimbaServer will run on the network server and when SimbaClient is installed on user machines, they will be able to access your database from general-purpose client applications.

5.1.4 Development Environments

C++

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

- Using the DSI API, build as an ODBC driver (connected locally to your data store) and link your DSII in this case upwards to SimbaODBC.
- Build as a SimbaServer driver (supporting remote connections from SimbaClients for JDBC, ADO.NET, and ODBC). You link your C++ DSII upwards to SimbaServer via the DSI API.

In either case, you may optionally use the DSI API Extensions linked against Simba SQLEngine to access non-relational data stores.

Java

For Java driver development, you have the following options:

- Using the Java DSI API, build as a JDBC driver (connected locally to your data store) and link your DSII in this case with SimbaJDBC.
- Build as an ODBC driver (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 driver (supporting remote connections from SimbaClients for JDBC, ADO.NET, and ODBC). You link your Java DSII upward via the Java DSI API and C++ to Java Bridge to SimbaServer.

When using the C++ to Java Bridge, you may optionally use the Java DSI API Extensions linked against Simba SQLEngine to access non-SQL-capable data stores.

C#

For C# development, you have the following options:

- Using the DotNet DSI API, build as an ADO.NET driver (connected locally to your data store) and link your DSII in this case with Simba.NET.
- Build as an ODBC driver (connected locally to your data store) using the DotNet DSI API and link via the C++ to C# Bridge to SimbaODBC.
- Build as a SimbaServer driver (supporting remote connections from SimbaClients for JDBC, ADO.NET, and ODBC). You link your DotNet DSII upward via the DotNet DSI API and C++ to C# Bridge to SimbaServer.

When using the C# to Java Bridge, you may optionally use the DotNet DSI API Extensions linked against Simba SQLEngine to access non-SQL-capable data stores.

5.2 Application Programming Interfaces

In Figure 11 and Figure 12, the interfaces labelled as ODBC API, JDBC API and ADO.NET API are industry standards that do not require further description in this document. See section 1.5, “For More Information” on page 9 for the location of the specifications for each of these.

All of the other interfaces are part of the SimbaEngine SDK. The detailed information about function parameters, return types, and error and message codes for all of these interfaces is available in the **SimbaEngine DSI API Reference Guide**, included as part of the SDK itself, available under the “\Documentation” folder.

This section provides high-level descriptions of significant classes to help you understand how the pieces fit together and what capabilities are available to you through the SimbaEngine SDK.

5.2.1 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 all the same functionality as the C++ classes.

The DSI API classes can be grouped as related to either the “Core” or the “Data Engine.” The Core classes provide all of the essential functionality to establish and manage the connection to your data source:

Class	Description
IDriver	The <code>IDriver</code> is a singleton instance constructed when the driver is first loaded. Its primary responsibility is to construct <code>IEnvironment</code> objects and manage any driver-wide properties. An abstract base class <code>DSIDriver</code> is provided to assist in some of these responsibilities; Most importantly, initializing defaults managing properties.
IEnvironment	The <code>IEnvironment</code> objects correspond to the ODBC environment (ENV) handles allocated by <code>SQLAllocHandle</code> . Their primary responsibility is to construct <code>IConnection</code> objects and manage any environment properties. An abstract base class <code>DSIEnvironment</code> is provided.
IConnection	The <code>IConnection</code> objects correspond to the ODBC connection (DBC) handles allocated by <code>SQLAllocHandle</code> . Their primary responsibility is to handle user authentication, construct <code>IStatement</code> objects, and manage any connection properties. An abstract base class <code>DSIConnection</code> is provided.
IStatement	The <code>IStatement</code> objects correspond to the ODBC statement (STMT) handles allocated by <code>SQLAllocHandle</code> . Their primary responsibility is to construct <code>IDataEngine</code> objects and manage any statement properties. An abstract base class <code>DSIStatement</code> is provided.
IMessageSource	The <code>IMessageSource</code> is responsible for loading error messages and

Class	Description
	warnings from your driver. An abstract implementation <code>DSIMessageSource</code> is provided to load messages generated by the SDK. See "Using or building a message source" in Appendix B: Errors and Exceptions on page 116 for more details.
<code>DSILog</code>	The <code>DSILog</code> is responsible for storing or printing log messages from your driver. Each of the <code>IDriver</code> , <code>IEnvironment</code> , <code>IConnection</code> , and <code>IStatement</code> classes has a <code>GetLog()</code> 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 <code>DSILog</code> 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 <code>ILogger</code> interface directly or subclassing <code>DSILog</code> .

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

Class	Description
<code>IDataEngine</code>	The <code>IDataEngine</code> is responsible for constructing an <code>IQueryExecutor</code> when preparing queries or constructing an <code>IResult</code> for catalog function metadata. An abstract base class <code>DSIDataEngine</code> is provided to assist in implementing filters for the catalog function metadata.
<code>IQueryExecutor</code>	The <code>IQueryExecutor</code> is responsible for executing a query and generating <code>IResult</code> objects.
<code>IResult</code>	An <code>IResult</code> 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 <code>DSISimpleResultSet</code> and <code>DSISimpleRowCountResult</code> are provided to deal with some basic functionality.

5.2.2 DSI API Extensions

These API Extensions provide access to the Simba SQL Engine through abstract or concrete implementation of the above data engine classes. The C# and Java versions of these classes, the DotNet DSI API Extensions and the Java DSI API Extensions, provide all the same functionality as the C++ classes.

Class	Description
<code>DSIExtSqlDataEngine</code>	The <code>DSIExtSqlDataEngine</code> is an abstract class, which derives from <code>DSIDataEngine</code> , is responsible for parsing and optimizing prepared SQL as well as opening tables from your data store.
<code>DSIExtQueryExecutor</code>	The <code>DSIExtQueryExecutor</code> is constructed by the <code>DSIExtSqlDataEngine</code> and is responsible for executing the query.
<code>DSIExtResultSet</code>	The <code>DSIExtResultSet</code> and the <code>DSIExtSimpleResultSet</code> are

Class	Description
	abstract classes derived from <code>IResult</code> . They require several new virtual functions, not required by <code>IResult</code> , to be implemented so that the result may be used in the SQL Engine. Tables opened by <code>DSIExtSqlDataEngine</code> must be instances of <code>DSIExtResultSet</code> .
DSIExtMetadataHelper	The <code>DSIExtMetadataHelper</code> is an optional abstract class that may be constructed by the <code>DSIExtSqlDataEngine</code> . It is responsible for iterating through tables and stored procedures so the engine can generate catalog function metadata.
DSIExtOperationHandlerFactory	The <code>DSIExtOperationHandlerFactory</code> is an optional abstract class that may be constructed by the <code>DSIExtSqlDataEngine</code> . It is responsible for constructing pass-down operation handlers that can optimize queries by allowing certain operations to be performed by your data store.

5.2.3 Lifecycle of DSI Objects

The objects of the DSI APIs in all three languages have the same lifecycles, loosely modeled on the lifecycle of ODBC handles.

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

The `IDriver` object creates `IEnvironments`, which are guaranteed to have been destroyed by the time the `IDriver` is destroyed.

`IEnvironments` create `IConnections`, which are guaranteed to have been destroyed by the time the parent `IEnvironment` has been destroyed. `IConnections` are typically long-lived objects, with multiple actions occurring before the `IConnection` is 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. Note, however, that `IDataEngines` are more of a factory object, and are typically destroyed after creating an `IQueryExecutor` or metadata result set.

`IDataEngines` create `IQueryExecutors`, which are not guaranteed to have been destroyed by the time the parent `IDataEngine` has been destroyed. Instead, they are guaranteed to have been destroyed by the time the parent statement has been destroyed. `IQueryExecutors` have a lifespan that matches the lifespan of a prepared and executed, or directly executed, query. A single `IQueryExecutor` is for multiple executions of a prepared query.

Note that if you are using the Simba `SQLEngine`, the `IQueryExecutor` is already implemented by the `SQLEngine`.

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

5.3 More depth on some of the basic features

The Build a Driver in 5 Days documents (each of the C++, C# and Java versions) walk you through the process of modifying the sample drivers to perform basic Data Retrieval and fetching of Metadata against your own data store. The Figure below shows the common design pattern discussed in that document.

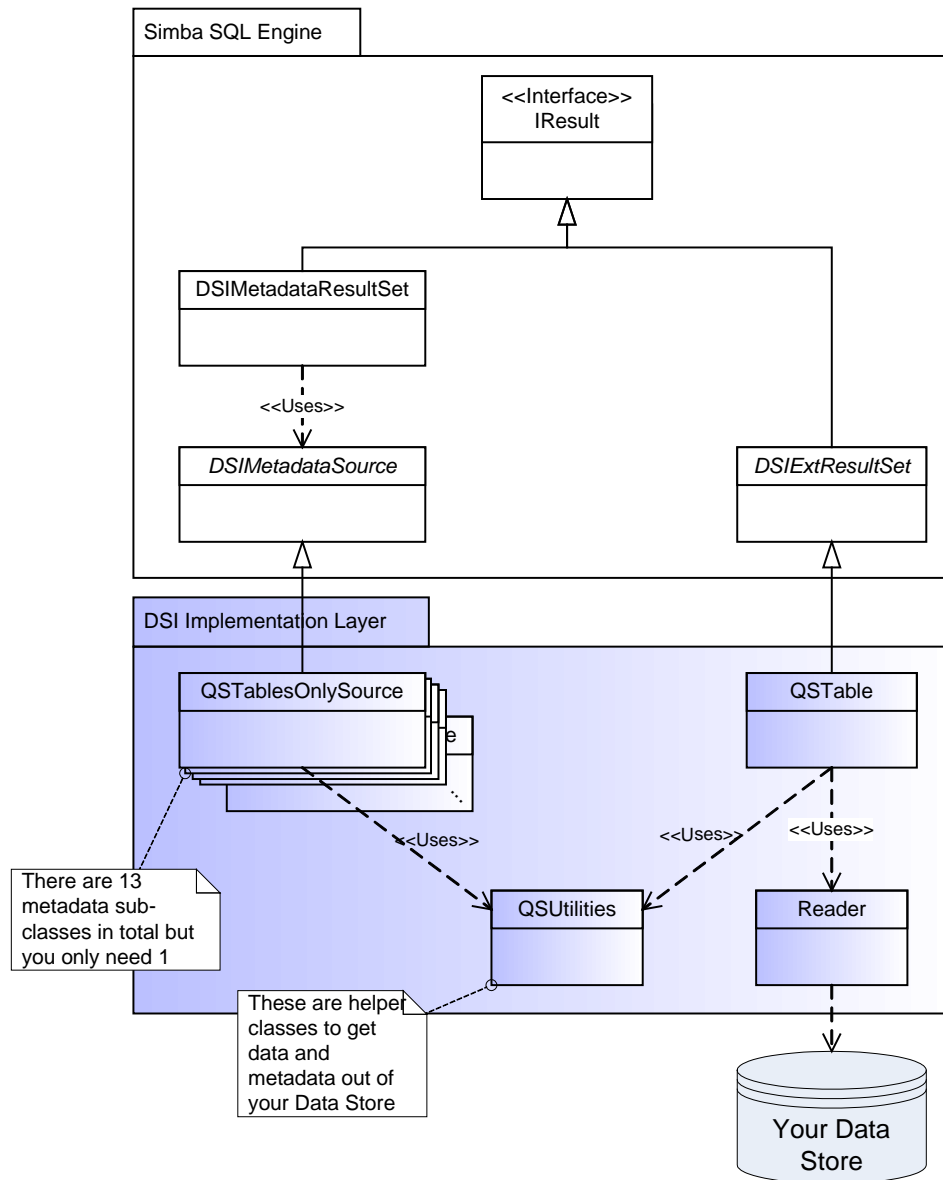


Figure 14: Design pattern for a DSI implementation.

5.3.1 Connection Parameters

The SimbaODBC component will pass all of the connection parameters that are defined in the connection string as well as those defined in the DSN entry (if one is used) for the connection. If entries of the connection string overlap entries from the DSN, then the connection string will override parameters from the DSN.

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

See Appendix D: The Connection Process for more information.

5.3.2 Data Retrieval

Appendix C: Data Retrieval in the Build a Driver in 5 Days documents (each of the C++, C# and Java versions) describes two methods to retrieve data from your data store. In that appendix, we note that you need to call the `GetBuffer()` method to get data into the buffer managed by Simba SQLEngine. An example of this, if the `SqlData` was of type `SQL_INTEGER` is:

```
simba_int32 value = 5;
memcpy(in_data->GetBuffer(), &value, sizeof(simba_int32));
```

or

```
*reinterpret_cast<simba_int32*>(in_data->GetBuffer()) = 5;
```

Either will work.

When working with variable length data (`SQL_CHAR/VARCHAR/LONGVARCHAR`, `SQL_WCHAR/WVARCHAR/WLONGVARCHAR`, `SQL_BINARY/VARBINARY/LONGVARBINARY`) you need to call `in_data->SetLength()` with the number of bytes that will be copied before using `memcpy()`.

5.3.3 Fetching Metadata

Of the 13 Metadata sub-classes, there is only one that you need to modify to make a basic driver work. This section describes the other metadata classes and under what circumstances you will need to update them. The implementation depends on whether the driver is using Simba SQLEngine or not.

Driver without Simba SQL Engine

If the driver is not using Simba SQL Engine, then the `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 driver 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.
- `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 driver 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.
- `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 driver, then a new class should be implemented by deriving from `Simba::DSI::DSIMetadataSource` and implementing all the functions.

NOTE: The Ultralight driver is a sample driver that does not use Simba `SQLEngine` and 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.

Driver with Simba SQL Engine

If the driver is using Simba SQL Engine, then the `CustomerDSIIDataEngine` class has to derive from `DSIExtSqlDataEngine`, and implement the following function:

```
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 `DSIMetadataSource*` which contains raw metadata. If the driver 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.
- `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 driver, then a new class should be implemented by deriving from `Simba::DSI::DSIMetadataSource` and implementing all the functions.

The driver is required to implement a class for `DSI_TYPE_INFO_METADATA` metadata table identifier, which is for catalog function `SQLGetTypeInfo`. This class should derive from pure abstract class called `DSIExtTypeInfoMetaDataSource` that has the following pure virtual function:

```
virtual Simba::SQLEngine::TypePrepared PrepareType(
    Simba::SQLEngine::SqlTypeInfo& io_typeInfo) = 0;
```

This function takes the specified SQL type information, modifies any fields that need to be changed to fit the data source, and indicates if that type is supported or not. For a sample implementation, you can look at Quickstart or Codebase sample driver.

The `SQLEngine` also provides default implementation for `DSI_TABLES_METADATA`, `DSI_CATALOGONLY_METADATA`, `DSI_SCHEMAONLY_METADATA`, `DSI_TABLETYPEONLY_METADATA`, `DSI_COLUMNS_METADATA`, `DSI_PROCEDURES_METADATA`, `DSI_PROCEDURES_COLUMNS_METADATA`, and `DSI_STATISTICS_METADATA` metadata table identifiers. If the driver chooses to use these default implementations, then the driver has to implement a class that derives from `Simba::SQLEngine::DSIExtMetadataHelper` and implement all the functions. The two pure virtual functions `GetNextProcedure()` and `GetNextTable()` are called by the default

implementations to retrieve the next procedure and the next table, respectively. For a sample implementaiton of `MetadataHelper` class, please refer to the Quickstart sample driver.

Note: If the driver does not use the default implementations for the metadata table identifiers mentioned above, then the driver should implement its own class for the metadata table identifiers by deriving from `Simba::DSI::DSIMetadataSource`. The driver should create an empty metadata source with no rows by returning a `DSIEmptyMetadataSource` object for the metadata table identifiers that it does not support. For sample implementation of `DSIMetadataSource` classes, please refer to the Codebase sample driver.

The table below list all the possible metadata table identifiers and what the metadata table is supposed to return:

DSIMetadataTableID	Description
DSI_TABLES_METADATA	Table containing table metadata of tables in the data source.
DSI_CATALOGONLY_METADATA	Table containing list of catalogs in the data source.
DSI_SCHEMAONLY_METADATA	Table containing list of schemas in the data source.
DSI_TABLETYPEONLY_METADATA	Table containing list of table types in the data source.
DSI_TABLE_PRIVILEGES_METADATA	Table containing metadata for the privileges tables defined in the data source.
DSI_COLUMNS_METADATA	Table containing metadata for the columns defined in the data source.
DSI_COLUMN_PRIVILEGES_METADATA	Table containing metadata for the column privileges defined in the data source.
DSI_FOREIGN_KEYS_METADATA	Table containing foreign key metadata for tables defined in the data source.
DSI_PRIMARY_KEYS_METADATA	Table containing primary key metadata for tables defined in the data source.
DSI_SPECIAL_COLUMNS_METADATA	Table containing metadata for the special columns defined in the data source.
DSI_STATISTICS_METADATA	Table containing metadata for the statistics for tables defined in the data source.
DSI_PROCEDURES_METADATA	Table containing metadata of procedures defined in the data source.
DSI_PROCEDURES_COLUMNS_METADATA	Table containing metadata for the columns of procedures in the data source.
DSI_TYPE_INFO_METADATA	Table containing metadata for the types defined in the data source.
DSI_CATALOGSCHEMAONLY_METADATA	Table containing list of schemas with catalogs in the data source.

5.3.4 Add Custom Metadata Columns to your Metadata Results

Custom metadata columns can be added to Metadata result tables. Custom columns allow Metadata results to incorporate 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]/Include/DSI/Client/DSICustomMetadataColumn.h`

Below is a sample implementation of a custom metadata column for `CustomerDSITablesMetadataSource`. Adding custom metadata columns to any other metadata source follows a similar formula.

- a. Define a custom column tag for the custom column

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

- b. Define a member variable for the custom column(s)

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

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

```
using namespace Simba::DataStoreInterface::DataEngine;  
using namespace Simba::Support;  
  
Client::DSICustomMetadataColumn* column = NULL;  
Client::DSIColumnMetadata* colMetadata = NULL;  
TypedDataWrapper::SqlTypeMetadata* metadata = NULL;  
  
// Custom column  
colMetadata = new Client::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 = Interface::DSI_NULLABLE;  
colMetadata->m_searchable = Interface::DSI_PRED_NONE;  
colMetadata->m_updatable = Interface::DSI_READ_ONLY;
```

```
// Create SqlTypeMetadata*
metadata = TypedDataWrapper::SqlTypeMetadataFactory::
    MakeNewSqlTypeMetadata(SQL_VARCHAR);
column = new Client::DSICustomMetadataColumn(
    metadata,
    colMetadata,
    CUSTOM_TABLES_COLUMN_TAG);
m_customColumnMetadata.push_back(column);
```

Please refer to the API Reference Guide for details about DSIColumnMetadata.

d. Implement CustomerDSIITablesMetadataSource::GetCustomColumns

```
void CustomerDSIITablesMetadataSource::GetCustomColumns(
    std::vector<Client::DSICustomMetadataColumn*>& out_customColumns)
```

e. Iterate over m_customColumns and push them into out_customColumns.

f. Implement CustomerDSIITablesMetadataSource::GetCustomMetadata

```
bool CustomerDSIITablesMetadataSource::GetCustomMetadata(
    simba_uint16 in_columnTag,
    TypedDataWrapper::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.
    }
}
```

5.4 Advanced Features

5.4.1 Collaborative Query Execution

Simba SQL Engine has features that allow a DSI to alter and optimize execution of a query according to the strengths of the data source. This takes place by providing access to the AE-

Tree (Algebraic Expression Tree) which is an object-oriented representation of the operations necessary to perform the query. The ability to optimize the tree comes in two different forms:

- Full access to the AE-Tree in which you can analyze it then add, remove, or alter the nodes.

Or

- Simba SQL Engine can analyze the tree for you and use pass-down handlers for the operations that can often be executed in the data store, thus eliminating the need for them to be processed in the SQL engine.

The SimbaEngine Codebase sample implements some Collaborative Query Execution optimizations and can be used as a reference for implementing your own optimizations.

The advantage of using Collaborative Query Execution is that it allows your DSII to take over execution of the parts of the SQL query that your data store excels at, while leaving the rest to the Simba SQL Engine. For instance, if your data store can join tables extremely quickly, then this operation can be executed by your DSII while Simba SQL Engine takes care of the rest of the operations. When Simba SQL Engine and your DSII use Collaborative Query Execution, your driver supports all of the SQL that Simba SQL Engine supports, while still exposing the strengths of your data store.

If your data store cannot perform any additional operations, then Collaborative Query Execution does not need to be used. Simba SQL Engine will still support the full range of SQL in a fast and efficient manner.

Algebraic Expression Tree

Before it executes an SQL statement, Simba SQL Engine can pass to your DSI implementation an optimized representation of the SQL statement, called an Algebraic Expression Tree, or AE-Tree. The SQL statement takes this form just before Simba SQL Engine transforms it into an execution plan and executes it. When Simba SQL Engine passes the AE-Tree to your DSI implementation, the DSI implementation can choose to execute any part of the AE-Tree itself. It signals its intentions by modifying the AE-Tree before returning it to Simba SQL Engine. For instance, if the data store can filter data, join data, or execute aggregate functions particularly fast, it can modify those nodes of the AE-Tree to point to the DSI implementation for execution. The DSI implementation can modify any part of the AE-Tree if it can perform the execution quickly, or it can replace the entire tree and execute the whole query itself.

There are four main types of AENode in an AE Tree: Statements; Boolean; Query Operations and Relational Expressions; and Values. Each of these is expanded on below; note that because your options in this area are potentially unlimited, we are focusing on the concepts at a high level in this document.

Statements:

The root node of the tree representing the type of operation being performed: Query, Procedure Call, or DML.

Boolean:

A logical expression representing a true or false outcome. The most common use of this is for the `WHERE` clause of a `SELECT` statement.

- The base class of this type of node is `AEBooleanExpr`.

Query Operations and Relational Expressions:

A representation of retrieval or manipulation of relational data such as selecting from a table.

- The base class of this type of node is `AEQueryOperation`. These nodes represent operations on the entire query result, such as sorting.
- `AERelationalExpr`, which derives from `AEQueryOperation`, is the base class of most other nodes of this type. They represent retrieval, filtering, or modification of some relational data. Typically, they take as an operand, one or two other relational expressions.
- Three such examples are:
 - i. `AETable` – This represents the retrieval of data from a table in your data store.
 - ii. `AESelect` – This represents the `WHERE` clause of a query by taking another relational expression and combining it with a boolean expression to use as a filter. The operand may be as simple as an `AETable` or a complicated combination of many other relational expressions.
 - iii. `AEJoin` – This represents any join of two tables or relational expressions and an optional boolean expression to use as a join condition.

Values:

A representation of a numeric or string value that either is a literal, parameter, or composed from an arithmetic expression of one or more other values.

- The base class of this type of node is `AEValueExpr`.
- The basic value expression nodes from which other values are built from are `AELiteral`, `AEColumn`, and `AEParameter`.

- Most arithmetic value expressions derive from `AEUnaryValueExpr` or `AEBinaryValueExpr` to represent an operation on one or two value expression operands.

Pass-Down Operation Handlers

After parsing a SQL query and generating the AE-Tree, Simba SQL Engine does analysis to identify three types of operations that a data store may be able to handle in an optimized way: Filters, Joins, and Aggregations. Upon identifying these, it uses an operation handler to pass down details of the operation to the DSII so that it may choose to fully or partially perform that operation. Or, if it can't perform the operation, it may leave it up to Simba SQL Engine to perform.

The operation handlers are constructed by a factory class, `DSIExtOperationHandlerFactory`, which you must subclass to construct your own handler classes. The factory class itself is to be constructed by overriding the `DSIExtSqlDataEngine::CreateOperationHandlerFactory` method.

Filter and Join Handlers:

Both filters and join pass-downs are handled by classes of type `IBooleanExprHandler`. They are constructed from one or two base tables and then the filter or join conditions are passed down. For each passed-down condition, the handler must return true or false indicating if it will process the filter. If it returns true, the SQL Engine will remove that filter clause from the AE-Tree and instead use the result set returned by `IBooleanExprHandler::TakeResult` instead of the base table(s). In the case of a filter handler, the returned result set must have the same columns as the base table. For a join handler, the returned result set must have all the columns of the left base table followed by all the columns of the right base table.

If the handler returns false there are several possibilities. If the filter or join condition can be broken down into two or more conjuncts in conjunctive normal form (CNF), the engine will attempt to pass down each conjunct individually, allowing your handler to choose true or false for each clause. Like before, for each clause that returns true when passed down, it will be removed from the AE-Tree. Clauses that return false will still be processed by the Simba SQL Engine.

If all clauses return false when passed down the engine, it will still call `IBooleanExprHandler::TakeResult`. If a non-null result set is returned, the engine will replace the base table(s) in the AE-Tree with the returned result but still process all the filter conditions on top of the new result. This allows a handler to partially process a filter in the data store, thus reducing the size of the result that the engine must fully process the filter on. If `TakeResult` returns null, then no change to the AE-Tree will be made.

To implement your handler there are several base classes to choose from: `IBooleanExprHandler`, `DSIExtAbstractBooleanExprHandler`, and

`DSIExtSimpleBooleanExprHandler`. `IBooleanExprHandler` just has one `Passdown` function to which all `AEBooleanExpr` nodes will be passed.

`DSIExtAbstractBooleanExprHandler` implements the `Passdown` function to delegate the passdown of each type of node to separate passdown functions.

`DSIExtSimpleBooleanExprHandler` subclasses further to implement several of these previous functions to identify and pass down some simple cases that are easier to handle, such as simple comparisons between two columns or a column and a literal.

Aggregation Handlers:

Passed down aggregations are handled by classes of the type `IAggregationHandler`. This base class defines one `Passdown` function that accepts an `AEAggregate` node. If the aggregation can be handled, the `Passdown` function must return a new result set that will represent the aggregation of the base result. If the aggregation cannot be handled, the `Passdown` function must return null and the SQL Engine will handle the aggregation.

To simplify analysis of the `AEAggregate` node, two abstract subclasses are defined.

`DSIExtAbstractAggregationHandler` divides the `AEAggregate` into passdowns for the individual aggregations needed and the individual groupings to use for the aggregations. If each aggregation and grouping passed down is accepted by returning true, the `CreateResult` function will be called to create the aggregation result that will replace the `AEAggregate` node in the AE-Tree. If any aggregation function or grouping passed down is rejected by returning false, the entire aggregate passdown will be abandoned and `CreateResult` will not be called.

The `DSIExtSimpleAggregationHandler` class derives further from `DSIExtAbstractAggregationHandler` to identify and pass down several simple cases that are easier to handle. It only passes down aggregations of literals or column references and only passes down groupings of column references. If the aggregation or grouping contains any more complex value expressions then the passdown will be rejected.

Unlike filter and join handlers, the columns in the generated result set do not match columns in the base table(s) directly. Instead, one column must be created for each aggregate function or grouping expression passed down. The order of the columns must match the order that the pass down functions are called in. That is, if `SetGroupingExpr` is called twice followed by `SetAggregateFn` once, the generated result set must contain exactly three columns; the first two being columns for the grouping values, the third being the aggregated result values.

Combining Pass-Downs:

For complex queries involving multiple tables, filters, joins, or aggregations, there may be many operations that can be passed down. However, to pass down an operation higher in the AE-Tree, all operations below that node must have been fully passed down. This limitation is required so that when constructing the new operation handler, the base tables passed to the

factory are always tables either opened by the data engine or constructed by earlier operation handlers. Therefore, if a table filter could not be passed down, the engine cannot pass down a join higher up in the AE-Tree involving the result of the table filter because the engine must first process the filter itself before the join can be performed.

Analyzing the AE-Tree:

After the query has been parsed, turned into an AE-Tree, and optimized (including pass-down optimizations) the DSII has an option to analyze the tree and optimize or alter it further. Doing this can completely change the query and is only recommended as a last resort if the optimizations cannot be performed elsewhere. This can be done by overriding the `DSIExtSqlDataEngine::CreateQueryExecutor` to intercept the `AEStatements` before calling the base class function to finish constructing the query executor:

```
DSIExtQueryExecutor* CustomerDSIISqlDataEngine::CreateQueryExecutor(
    AutoPtr<AEStatements> in_aeStatements)
{
    // Analyze and alter the AE-Trees found in in_aeStatements
    return DSIExtSqlDataEngine::CreateQueryExecutor(in_aeStatements);
}
```

How to analyze the tree, and what changes to make, is up to the DSII implementer according to whatever circumstances require it so no example can be given here. Instead, refer to the API reference guide for full details on the structure of the AE-Tree.

5.4.2 Stored Procedures

An application can call a procedure instead of a SQL statement. A stored procedure is an operation that allows the DSI to take advantage of internal functions or extended non-SQL functionality that may exist. Custom functions that can be implemented inside these stored procedures may allow access to data that is not stored in standard relational tables.

SimbaEngine provides the following DSI API classes to support stored procedures:

- `DSIExtProcedure` – The base class for DSII stored procedures. Implement the pure virtual functions to provide functionality.
- `DSIExtMetadataHelper` – for `GetNextProcedure()`. This is optional; the Codebase example doesn't use this and returns them manually.
 - If `GetNextProcedure()` is not implemented, the DSII will need to implement the metadata sources for `SQLProcedures` and `SQLProcedureColumns`.
- `DSIExtSqlDataEngine` – to implement `OpenProcedure()`, same as `OpenTable()`.

The key class here is `DSIExtProcedure`, which provides the framework for custom stored procedures. These are returned to `SQLEngine` via `DSIExtSqlDataEngine::OpenProcedure()`.

Codebase supplies many examples of all the different ways that stored procedures can be used, and is an excellent reference.

5.4.3 DML

Simba SQL Engine can support SQL DML (Data Manipulation Language) queries if your data source supports it. This includes DELETE, INSERT, and UPDATE queries. To implement this, the following changes and additions must be made to your code:

1. Set the `DSI_CONN_ACCESS_MODE` and `DSI_CONN_DATA_SOURCE_READ_ONLY` properties in your `CustomerDSIIConnection` object. This can be done using the `DSIPropertyUtilities::SetReadOnly` helper method.
2. Modify your `DSIExtSqlDataEngine::OpenTable` method to accept an `in_openType` value of `TABLE_OPEN_READ_WRITE` and open the table appropriately.
3. For any table returned by `OpenTable` in `TABLE_OPEN_READ_WRITE` mode, the following virtual methods from `DSIExtResultSet` must be overridden and implemented:
 - a. `AppendRow()` – Appends an empty row the end of the result set and positions the cursor on the new row.
 - b. `DeleteRow()` – Deletes the current row from the result set.
 - c. `WriteData()` – Writes data to a column in the current row of the result set.

See the API Reference Guide for full details on these methods.

1. The following virtual methods from `DSIExtResultSet` may also optionally be overridden and implemented:
 - a. `OnStartRowUpdate()` – Called before writing data to update a row. This is not called after `AppendRow` as it is implied data will be written.
 - b. `OnFinishRowUpdate()` – Called after writing all updated or inserted data in a row.

5.4.4 Transactions

To enable transactions, your `DSII` must be write-capable. Once done, you can enable support for beginning, committing, and rolling back transactions by making the following changes to your code:

2. Set the `DSI_CONN_TXN_CAPABLE` property in your `CustomerDSIIConnection` object to the appropriate level of transaction support. This can be done using `DSIPropertyUtilities::SetTransactionSupport` helper method.
3. Modify your `CustomerDSIIConnection` object to override and implement the following virtual methods:
 - a. `BeginTransaction()` – Begins a new transaction on the connection.
 - b. `Commit()` – Commits any outstanding transaction on the connection.
 - c. `Rollback()` – Roll back any outstanding transaction on the connection.

If your DSII is written in Java and is using the SimbaJDBC component, then you may optionally implement Savepoints as well.

1. Set the `DSI_SUPPORTS_SAVEPOINTS` property in your `CustomerDSIIConnection` object to `DSI_SUPPORTS_SAVEPOINTS_TRUE`.
2. Modify your `CustomerDSIIConnection` object to override and implement the following virtual methods:
 - a. `createSavepoint(String)` – Creates a new Savepoint with the specified name in the current transaction.
 - b. `releaseSavepoint(String)` – Releases the Savepoint with the specified name.
 - c. `rollback(String)` – Rolls back the transaction to the Savepoint with the specified name.

Supporting Transactions through SQL

In some data sources, transactions can also be triggered by executing certain SQL queries. Support for this is provided through the `ITransactionStateListener` interface, allowing your DSII to inform the Simba components of any changes in transaction state. To implement this, the following changes and additions must be made to your code:

1. Optionally override and implement the `registerTransactionStateListener` virtual method in your `CustomerDSIIConnection` object. If you do not override this method, you can still access the registered transaction listener through a member variable in the `DSIConnection` object.
2. Inform the Simba components of an changes to transaction state:
 - a. When a transaction has started, call `ITransactionStateListener::NotifyBegin`
 - b. When a transaction is committed, call `ITransactionStateListener::NotifyCommit`.

- c. When a transaction is rolled back, call `ITransactionStateListener::NotifyRollback`.

If your DSI 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`.

Note that ODBC does not support Savepoints, and attempting to use the `notify*Savepoint` functions on the `ITransactionStateListener` while using the JNI DSI API will result in an exception.

5.4.5 ODBC Custom Data Types

SimbaEngine SDK allows DSI implementers to add additional data types based on existing types, allowing for custom behavior either with or without the `SQLEngine`. Each data type that is added must be based on an existing data type so that it can be exposed to applications without custom logic in those applications. For instance, a Money type might be added that is based on Numeric, but is restricted to two decimal places, perhaps with a custom conversion to character types that adds the currency character.

SimbaEngine SDK will use a `UtilityFactory` class to create a `SqlTypeMetadataFactory` object to create the metadata about the custom types, a `SqlDataFactory` to create the object which represents the custom type, and a `SqlConverterFactory` to convert the custom type to other data types.

To add this functionality to your ODBC driver, make the following changes:

1. Modify your `CustomerDSIIDriver` object to override and implement the virtual method `CreateUtilityFactory()` to return a `CustomerDSIIUtilityFactoryClass`. This class will provide the other factories that implement custom data type behavior.
2. Create a `CustomerDSIIUtilityFactory` class which 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. `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 `CustomerDSIISqlTypeMetadataFactory` 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 SQL data types. 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.
 4. Create a `CustomerDSIISqlDataFactory` 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 if supported, otherwise return `NULL`.
 5. Create a `CustomerDSIISqlConverterFactory` class which subclasses `Simba::Support::SqlConverterFactory`, and override and implement the following virtual methods:
 - a. `CreateNewSqlToCConverter()` – Takes a `SqlData` and `SqlCData` object representing the source and target types, and an `IWarningListener` for posting any conversion warnings to. The returned converter is responsible for converting from the source SQL data type to the target C data type.
 - b. `CreateNewCToSqlConverter()` – Takes a `SqlCData` and `SqlData` object representing the source and target types, and an `IWarningListener` for posting any conversion warnings to. The returned converter is responsible for converting from the source C data type to the target SQL data type.
 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.

If your driver is using the Simba `SQLEngine`, then you can customize the behavior of the data types within the `SQLEngine` as well by making the following changes:

7. In the `CustomerDSIISqlConverterFactory` class, override and implement the following virtual methods:
 - a. `CreateNewSqlToSqlConverter()` – Takes a `SqlData` and `SqlData` object representing the source and target types, and an `IWarningListener` for posting any conversion warnings to. The returned converter is responsible for converting from the source SQL data type to the target SQL data type.
8. Modify your `CustomerDSIIDataEngine` object to override and implement the virtual method `CreateBehaviorProvider()` to return a `CustomerDSIIBehaviorProvider`. This class will provide the other factories that provide the custom data type `SQLEngine` behavior.
9. Create a `CustomerDSIIBehaviorProvider` class which subclasses `Simba::SQLEngine::DSIExtCustomBehaviorProvider`. This factory class will initialize classes that handle the type coercion and custom type behavior within the `SQLEngine`.
 - a. `InitializeCellComparatorFactory()` – Initializes the `m_cellComparatorFactory` member with a `CustomerDSIICellComparatorFactory` that implements `ICellComparator` for comparing two data type values.
 - b. `InitializeCoercionHandler()` – Initializes the `m_coercionHandler` member with a `CustomerDSIICoercionHandler` that implements `ICoercionHandler` for coercing different data types into one data type.
 - c. `InitializeCollatorFactory()` – Initializes the `m_collatorFactory` member with a `CustomerDSIICollatorFactory` that implements `ICollatorFactory` for comparing/collating text.
 - d. `InitializeFunctorFactory()` – Initializes the `m_functorFactory` member with a `CustomerDSIIFunctorFactory` that implements `IFunctorFactory` for returning functors that perform operations on specific data types.

Note that the behavior provider does not need to override all of the functions, it need only override the functions that provide the functionality that needs to be customized.

10. Create a `CustomerDSIICellComparatorFactory` class which implements `ICellComparatorFactory` if custom comparisons are needed. The factory should override and implement the following virtual functions:
 - a. `MakeNewCellComparator()` – Create a cell comparator that can compare two values of the type specified by the passed in `SqlTypeMetadata`. The cell comparator should be able to do comparisons such that it can determine when two values are equal, or one is greater than the other. Return `NULL` if the comparison is not supported.

11. Create a `CustomerDSIICoercionHandler` class which extends `DSIExtCoercionHandler` if custom coercions between two types are needed. The handler should override and implement the following virtual functions:
 - a. `Coerce*Type()` – Takes `SqlTypeMetadata` objects and coerces them to create one result `SqlTypeMetadata` that would result from the operation that the metadata is being coerced for. Return `NULL` if the coercion is not supported.
 - b. `Coerce*ColumnMetadata()` – Takes `ColumnMetadata` objects and coerces them to create one result `ColumnMetadata` that would result from the operation that the metadata is being coerced for. Return `NULL` if the coercion is not supported.
12. Create a `CustomerDSIICollatorFactory` class which extends `DSIExtCollatorFactory` if custom collations are needed. The factory should override and implement the following virtual functions:
 - a. `GetDefaultCustomCollation()` – Return a collation if the default is not what is desired.
 - b. `GetCustomCollationByName()` – Return the ID for the specified collation name.
 - c. `CreateCustomCollator()` – Create a new `ICollator` that handles collation of the specified collation sequence. Return `NULL` if the collation is not supported.
13. Create a `CustomerDSIIFunctorFactory` class which extends `DSIExtFunctorFactory` if custom operation behavior is needed. The factory should override and implement the following virtual functions:
 - a. `CreateBinaryArithmeticFunctor()` – Create a new `DSIExtBinaryValueFunctor` subclass which handles the specified arithmetic operation of two values of the type specified by the `SqlTypeMetadata`. The `DSIExtBinaryValueFunctor` should operate on the `m_leftData` and `m_rightData` member values during `Execute()`.
 - b. `CreateComparisonFunctor()` – Create a new `DSIExtBinaryBooleanFunctor` subclass which handles the specified comparison between two values of the type specified by the `SqlTypeMetadata`. The `DSIExtBinaryBooleanFunctor` should operate on the `m_leftData` and `m_rightData` member values during `Execute()`, and use the `ICollator` supplied by the `ICollatorFactory` if needed.
 - c. `CreateNegationFunctor()` – Create a new `DSIExtUnaryValueFunctor` subclass which handles negation of the type specified by the `SqlTypeMetadata`. The `DSIExtUnaryValueFunctor` should operate on the `m_data` member during `Execute()`.
 - d. `CreateExistsFunctor()` – Create a new `DSIExtBinaryValueFunctor` subclass which handles the `EXISTS` clause for the type specified by the

`SqlTypeMetadata`. The `DSIExtBinaryValueFuncor` should operate on the `m_leftData` and `m_rightData` member values during `Execute()`.

- e. `CreateInFuncor()` – Create a new `DSIExtBinaryValueFuncor` subclass which handles the IN clause for the type specified by the `SqlTypeMetadata`. The `DSIExtBinaryValueFuncor` should operate on the `m_leftData` and `m_rightData` member values during `Execute()`.
- f. `CreateLikeFuncor()` – Create a new `DSIExtBinaryValueFuncor` subclass which handles the LIKE clause for the type specified by the `SqlTypeMetadata`. The `DSIExtBinaryValueFuncor` should operate on the `m_leftData` and `m_rightData` member values during `Execute()`.

5.4.6 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 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.

5.4.7 JDBC Updatable ResultSets

JDBC provides the functionality to modify result sets that are generated from statements. SimbaJDBC allows you to easily add this functionality to your JDBC driver 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` as it is implied data will be written.
 - b. `onFinishRowUpdate()` – Called after writing all updated or inserted data in a row.

5.5 Add Custom Connection & Statement Attributes

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

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 drivers 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, know that your driver may experience compatibility issues with any application that uses the conflicting custom attributes for other drivers.

Refer to the *Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostics Types, and Attributes* section of the MSDN ODBC Programmer's Reference.

You will need to 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, you will need to create a `Simba::Support::AttributeData*` to store data for the property or attribute. A map should be used to map keys to their corresponding `AttributeData*`. Please refer to the `DSIConnProperties.h` or `DSIStmtProperties.h` headers, which can be found at `<installdir>/Include/DSI` to see how this can be achieved.

Provide proper implementations for the following methods in your `CustomerDSIIConnection` and/or `CustomerDSIIStatement` class:

IsCustomProperty

This function should check if the provided key corresponds with one of the standard ODBC properties and return false if it does not; true otherwise.

The list of keys for ODBC properties can be found in the `Simba::DSI::DSISstmtPropertyKey` enum or the `Simba::DSI::DSIConnPropertyKey` enum.

SetCustomProperty

This function allows you to set an `AttributeData*` for the custom property key. Note in your implementation, you should check to ensure the provided key indeed corresponds to a custom property/attribute. If it does not, an appropriate error or exception should be thrown and logged (if applicable).

GetCustomProperty

This function retrieves the `AttributeData*` associated with a custom key. Note that in your implementation, you should check to ensure the provided key indeed corresponds to a custom property/attribute.

GetCustomPropertyType

This function retrieves the data type associated with the custom property/attribute. Data types are defined in the `Simba::Support::AttributeType` enum. The definition for this enum can be found in the `<installdir>/Include/Support/AttributeData.h` header file.

5.5.1 Java DSI Custom Properties

Custom properties can be added to the drivers 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 drivers, while when using the SimbaJDBC component the custom properties are exposed through custom extensions to the Connection and Statement object:

1. `getAttribute(int)` – Retrieve a custom property identified by the integer key.
2. `setAttribute(int, Object)` – Set a custom property identified by the integer key.

Note that since these are custom extensions, applications will have to be coded to explicitly use these functions.

5.5.2 DotNet DSI Custom Properties

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

5.6 Overriding Default Property and Attribute Values

Each property and attribute for the Core classes DSIEnvironment, DSIConnection, DSIDriver and DSISStatement is assigned a default value. These values are initialized at time of instantiation (i.e. when the constructor is called). A full listing of all properties and attributes and their default values can be found in the API Reference Guide.

If the capabilities of your driver differ from the default properties or a specific targeted application requires different values for any of the property or attribute values, the values can be overridden in the DSII subclasses of DSIEnvironment, DSIConnection, DSIDriver and DSISStatement.

The following details how to override default property and attribute values for the customer subclass implementation of DSIConnection (CustomerDSIConnection). The details for overriding default values for DSISStatement, DSIDriver and DSIEnvironment are the same.

Property and attributes are represented as key-value string pairs. These pairs are stored in a map. Each of these four Core classes has a map. The name of each property map is shown below:

Class	Name of property map
DSIConnection	m_connectionProperties
DSIDriver	m_driverProperties
DSIEnvironment	m_environmentProperties
DSISStatement	m_statementProperties

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

Note that the default values for any properties should only be overridden 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 **CustomerDSIConnection::Connect** function.

As an example, the steps to override DSIConnection properties will be discussed.

In the `CustomerDSIConnection` constructor, set the property or attribute keys to the desired values. Some examples are shown below:

The signature of the `DSIConnection::SetProperty` function is:

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

Example # 1: 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 # 2: 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, the default value will need to be changed. Here, the value for the `DSI_SUPPORTED_CHAR_CONVERSIONS` property is made up of a concatenation of all the values provided. See below.

```
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));
```

5.6.1 JDBC Specific Properties

These properties are specific to drivers that use the Java DSI along with SimbaJDBC to build a JDBC driver.

Driver Properties

- **DSI_UID_PWD_MAPPING** – Determines if SimbaJDBC should map from the JDBC standard of User/Password to the ODBC standard of UID/PWD. This may be useful if using the same DSI with both SimbaJDBC and JNI DSI to ensure that the same keys are used. Available values are:
 - **DSI_UID_PWD_MAPPING_TRUE** – SimbaJDBC maps User/Password to UID/PWD.
 - **DSI_UID_PWD_MAPPING_FALSE** – SimbaJDBC does not map User/Password to UID/PWD.

Defaults to **DSI_UID_PWD_MAPPING_FALSE**.

- **DSI_BATCH_STOP_ON_ERROR** – Determines if batch processing, via the JDBC batch execution API, should stop when it encounters the first error or should process all the results. Available values are:
 - **DSI_BATCH_STOP_ON_ERROR_TRUE** – Batch processing will stop when the first error is encountered.
 - **DSI_BATCH_STOP_ON_ERROR_FALSE** – Batch processing will not stop when an error is encountered, and will process all results.

Defaults to **DSI_BATCH_STOP_ON_ERROR_FALSE**.

- **DSI_SUPPORTS_UPDATE_BATCHING** – Determines if update batch processing via the JDBC batch execution API is supported. Note that the `prepareBatch()` method will need to be overridden and implemented if this property is enabled. Available values are:
 - **DSI_SUPPORTS_UPDATE_BATCHING_TRUE** – Indicates the driver supports update batching.
 - **DSI_SUPPORTS_UPDATE_BATCHING_FALSE** – Indicates the driver does not support update batching.

Defaults to **DSI_SUPPORTS_UPDATE_BATCHING_FALSE**.

- **DSI_SUPPORTS_NAMED_PARAMETERS** – Determines if named parameters are supported for `CallableStatement` objects. Available values are:

- `DSI_SUPPORTS_NAMED_PARAMETERS_TRUE` – Indicates the driver supports named parameters.
- `DSI_SUPPORTS_NAMED_PARAMETERS_FALSE` – Indicates the driver does not support named parameters.

Defaults to `DSI_SUPPORTS_NAMED_PARAMETERS_FALSE`.

- **`DSI_SUPPORTS_SELECT_FOR_UPDATE`** – Determines if the data source supports the “SELECT FOR UPDATE” SQL syntax. Available values are:
 - `DSI_SUPPORTS_SELECT_FOR_UPDATE_TRUE` – Indicates the data source supports the “SELECT FOR UPDATE” SQL syntax.
 - `DSI_SUPPORTS_SELECT_FOR_UPDATE_FALSE` – Indicates the data source does not support the “SELECT FOR UPDATE” SQL syntax.

Defaults to `DSI_SUPPORTS_SELECT_FOR_UPDATE_FALSE`.

- **`DSI_SUPPORTS_MULT_ACTIVE_RESULT_SETS`** – Determines if the driver supports multiple active result sets on a single statement. Available values are:
 - `DSI_SUPPORTS_MAR_TRUE` – Indicates the driver supports multiple active result sets on a single statement.
 - `DSI_SUPPORTS_MAR_FALSE` – Indicates the driver does not support multiple active result sets on a single statement.

Defaults to `DSI_SUPPORTS_MAR_FALSE`.

Connection Properties

- **`DSI_SUPPORTS_SAVEPOINTS`** – Determines if the data source supports Savepoints. Available values are:
 - `DSI_SUPPORTS_SAVEPOINTS_TRUE` – Indicates the data source supports Savepoints.
 - `DSI_SUPPORTS_SAVEPOINTS_FALSE` – Indicates the data source does not support Savepoints.

Defaults to `DSI_SUPPORTS_SAVEPOINTS_FALSE`.

- **`DSI_SUPPORT_UPDATABLE_RESULT_SETS`** – Determines the level of support for updatable result sets that the driver provides. The value is a bitmask of the available values:

- `DSI_SUPPORTS_URS_NONE` – Indicates no support for updatable result sets.
- `DSI_SUPPORTS_URS_DELETE` – Indicates support for deletion of rows in result sets.
- `DSI_SUPPORTS_URS_INSERT` – Indicates support for insertion of new rows in result sets.
- `DSI_SUPPORTS_URS_UPDATE` – Indicates support for updates of rows in result sets.

Defaults to `DSI_SUPPORTS_URS_NONE`.

- **`DSI_RESULT_SET_CHANGES_DETECTED`** – Determines the level of support for detecting changes in result sets. The value is a bitmask of the available values:
 - `DSI_RS_CD_NONE` – No changes are detected.
 - `DSI_RS_CD_DELETE_FORWARD` – Deletions are detected in forward-only result sets.
 - `DSI_RS_CD_INSERT_FORWARD` – Insertions are detected in forward-only result sets.
 - `DSI_RS_CD_UPDATE_FORWARD` – Updates are detected in forward-only result sets.
 - `DSI_RS_CD_DELETE_INSENSITIVE` – Deletions are detected in scroll insensitive result sets.
 - `DSI_RS_CD_INSERT_INSENSITIVE` – Insertions are detected in scroll insensitive result sets.
 - `DSI_RS_CD_UPDATE_INSENSITIVE` – Updates are detected in scroll insensitive result sets.
 - `DSI_RS_CD_DELETE_SENSITIVE` – Deletions are detected in scroll sensitive result sets.
 - `DSI_RS_CD_INSERT_SENSITIVE` – Insertions are detected in scroll sensitive result sets.
 - `DSI_RS_CD_UPDATE_SENSITIVE` – Updates are detected in scroll sensitive result sets.

Defaults to `DSI_RS_CD_NONE`.

5.6.2 ADO.NET Specific Properties

There are no properties that are specific to drivers that use the DotNet DSI along with Simba.NET to build an ADO.NET driver.

5.6.3 SQLEngine Specific Properties

These properties are specific to drivers that use the Simba SQLEngine, and can be set on the `DSIExtSqlDataEngine`.

- **DSIEXT_DATAENGINE_NULL_EQUALS_EMPTY_STRING** – Determines if the IS NULL predicate is treated the same way that = '' (empty string) is. Available values are:
 - “Y” – Indicates that IS NULL and = '' are treated the same way.
 - “N” – Indicates that IS NULL and = '' are not treated the same way.

Defaults to “N”.

- **DSIEXT_DATAENGINE_TEMPORARY_INDEXES** – Determines if the SQLEngine will create temporary indexes when performing joins when appropriate indexes are not available. Available values are:
 - “Y” – Indicates the SQLEngine will create temporary indexes for joins.
 - “N” – Indicates the SQLEngine will not create temporary indexes for joins.

Defaults to “Y”.

- **DSIEXT_DATAENGINE_TABLE_CACHING** – Determines if the SQLEngine will cache tables that it reads if a row would be visited more than once, so that the DSII can safely discard data once a row has been visited. Available values are:
 - “Y” – Indicates the SQLEngine will cache tables if needed.
 - “N” – Indicates the SQLEngine will not cache tables, and the DSII should cache the tables to allow rows to be visited multiple times.

Defaults to “N”.

SQLEngine Specific Switches

The SQLEngine also provides some switches to customize its behavior. These switches are read from either HKLM\SOFTWARE\\Driver in the Windows Registry or from the vendor .ini file on non-Windows platforms.

- **SwapFilePath** – Set the path that SQLEngine will use for creation of temporary swap files that are needed for certain operations.

6 Compiling your DSI

The Build a Driver in 5 Days documents (each of the C++, C# and Java versions) walk you through the basic steps to compile your driver, focusing primarily on the Windows platform. This section goes into more depth on the subject and the variations for other platforms.

6.1 C++ under Windows

The following are specifications for your Project Properties page in Microsoft Visual Studio 2008 or 2010. Please refer to the Microsoft MSDN documentation for a full listing of all compiler options.

6.1.1 Build as an ODBC Driver (a DLL) for local connections

The sample drivers discussed in the Build a C++ Driver in 5 Days document are set up to build as Windows DLL's. To build your own driver as a Windows DLL, confirm the following settings:

1. Set configuration type to `Dynamic Library (.dll)`:

Configuration Properties → General → Configuration Type

2. Link against `SimbaODBC_$(ConfigurationName).lib`:

Configuration Properties → Linker → Input → Additional Dependencies

ConfigurationName is one of these four values: `Debug`, `Debug_MTDLL`, `Release` or `Release_MTDLL`. See section 6.1.5, "Run-time library options" below for related settings for each option.

3. Set module definition file to `Exports.def` included in all sample drivers:

Configuration Properties → Linker → Input → Module Definition File

4. Set output file to a DLL name:

Configuration Properties → Linker → General → Output File

5. Include the DSI and Support include paths:

Configuration Properties → C/C++ → General → Additional Include Directories:

1. `$(SIMBAENGINE_DIR)\Include\DSI`
2. `$(SIMBAENGINE_DIR)\Include\DSI\Client`
3. `$(SIMBAENGINE_DIR)\Include\Support`
4. `$(SIMBAENGINE_DIR)\Include\Support\Exceptions`
5. `$(SIMBAENGINE_DIR)\Include\Support\TypedDataWrapper`

6.1.2 Build as a SimbaServer (an EXE) for remote connections

To build a driver as a stand-alone SimbaServer executable, the key settings needed are:

1. Set configuration type to `Application (.exe)`:

Configuration Properties → General → Configuration Type

2. Link against `SimbaServer_$(ConfigurationName).lib`:

Configuration Properties → Linker → Input → Additional Dependencies

ConfigurationName is one of these four values: `Debug`, `Debug_MTDLL`, `Release` or `Release_MTDLL`

See section 6.1.5, "Run-time library options" below for related settings for each option..

3. Unset module definition file:

Configuration Properties → Linker → Input → Module Definition File

4. Set output file to an EXE name:

Configuration Properties → Linker → General → Output File

5. Include the DSI and Support include paths:

Configuration Properties → C/C++ → General → Additional Include Directories:

1. `$(SIMBAENGINE_DIR)\Include\DSI`
2. `$(SIMBAENGINE_DIR)\Include\DSI\Client`
3. `$(SIMBAENGINE_DIR)\Include\Support`
4. `$(SIMBAENGINE_DIR)\Include\Support\Exceptions`
5. `$(SIMBAENGINE_DIR)\Include\Support\TypedDataWrapper`

6.1.3 Building with Simba SQLEngine

The SimbaEngine Quickstart and Codebase sample drivers demonstrate connections to non-SQL data sources using Simba SQLEngine.

1. All settings apply from either 6.1.1 or 6.1.2 above, depending on ODBC Driver or SimbaServer configuration.
2. Additionally link against `SimbaEngine_$(ConfigurationName).lib`:

Configuration Properties → Linker → Input → Additional Dependencies

ConfigurationName is one of these four values: `Debug`, `Debug_MTDLL`, `Release` or `Release_MTDLL`

See section 6.1.5, "Run-time library options" below for related settings for each option.

3. Include the SQLEngine include paths:

Configuration Properties → C/C++ → General → Additional Include Directories:

1. `$(SIMBAENGINE_DIR)\Include\SQLEngine`
2. `$(SIMBAENGINE_DIR)\Include\SQLEngine\AETree`
3. `$(SIMBAENGINE_DIR)\Include\SQLEngine\DSIExt`

6.1.4 Building without Simba SQLEngine

The SimbaEngine Ultralight sample driver demonstrates a connection to a data source that provides SQL processing.

1. All settings apply from either 6.1.1 or 6.1.2 above, depending on ODBC Driver or SimbaServer configuration.
2. No additional libraries need to be linked.
3. No additional paths need to be included.

6.1.5 Run-time library options

When linking, there are 4 versions of each Simba library file with different configuration names (Debug, Debug_MTDLL, Release and Release_MTDLL). In order to link against each, your project settings must match certain settings used to build the libraries:

Debug

Debug Simba libraries using statically linked C++ runtime.

Configuration Properties → C/C++ → Preprocessor → Preprocessor Definitions:

- Must include: `_DEBUG`

Configuration Properties → C/C++ → Code Generation → Runtime Library:

- Multi-threaded Debug (/MTd)

Debug_MTDLL

Debug Simba libraries using dynamically linked C++ runtime.

Configuration Properties → C/C++ → Preprocessor → Preprocessor Definitions:

- Must include: `_DEBUG`

Configuration Properties → C/C++ → Code Generation → Runtime Library:

- Multi-threaded Debug DLL (/MDd)

Release

Release Simba libraries using dynamically linked C++ runtime.

Configuration Properties → C/C++ → Preprocessor → Preprocessor Definitions:

- Must include: `NDEBUG`

Configuration Properties → C/C++ → Code Generation → Runtime Library:

- Multi-threaded (/MT)

Release_MTDLL

Release Simba libraries using dynamically linked C++ runtime.

Configuration Properties → C/C++ → Preprocessor → Preprocessor Definitions:

- Must include: `NDEBUG`

Configuration Properties → C/C++ → Code Generation → Runtime Library:

- Multi-threaded DLL (/MD)

6.1.6 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” (if you based your project on one of the sample drivers, it should be set that way by default).

6.2 C# under Windows

Building an ADO.NET provider using Simba.NET is the main way of using C# .Net, however the same four build types described above (ODBC or server, with or without Simba SQL Engine) can also be developed using C# .Net. Most of the settings described for C++ still apply and the main difference is the addition of a new project for the .Net DSII code.

6.2.1 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. Instead, you add references to the Simba.DotNetDSI and Simba.DotNetDSIExt assemblies (If you are not using the Simba SQL Engine, only the Simba.DotNetDSI reference is necessary). The base classes from which you derive to code your DSII are all defined in these two assemblies. These two assemblies are bit-agnostic, and can be used for both 32-bit and 64-bit driver development. Note that if you are using the CLIDSII, you will need to set the bitness of your compiled C# DLL to match the bitness of the CLIDSII library that is being used.

For the purposes of examples, the output of this project will be referred to as CustomerDotNetDSII.dll. When using your provider, server, or ODBC driver, this assembly should be registered in the Global Assembly Cache (GAC) of Windows.

6.2.2 Simba.NET

In order to build a pure C# ADO.NET provider, you will only need your DotNet DSII project, and the Simba.DotNetDSI and Simba.ADO.NET assemblies. Note that you cannot use the Simba SQL Engine when building a pure C# ADO.NET provider and must use SimbaServer and one of the clients to do so.

When using Simba.NET to create an ADO.NET provider, you will also need to extend a few additional abstract classes that are not part of the normal DotNet DSI. These are part of the Simba.ADO.NET assembly and are listed below:

- SCommand

- SCommandBuilder
- SConnection
- SConnectionStringBuilder
- SDataAdapter
- SFactory
- SParameter

In most cases no additional code is needed, with the one exception being the `SConnectionStringBuilder` subclass. Here you should add any additional properties that are needed to establish a connection to your provider. An example is available in the Simba DotNetUltralight sample.

When using your provider, the `Simba.DotNetDSI` and `Simba.ADO.NET` assemblies should be registered in the Global Assembly Cache (GAC) of Windows.

6.2.3 Build as an ODBC Driver (a DLL) for local connections

In order to build an ODBC driver with or without Simba SQL Engine, in addition to the DotNet DSI project, you must create a CLDSI project that provides the bridge between the native C++ Simba ODBC libraries and your `CustomerDotNetDSII.dll` assembly. This is done by creating a project following the instructions above for C++ under Windows.

In addition to the library files listed in either 6.1.1 or 6.1.3 above, you must also include `CLDSI_$(ConfigurationName).lib`. This library forms the bridge between unmanaged DSI classes to the managed DSI classes. The other configuration change you must make is to enable Common Language Runtime Support (/clr):

Configuration Properties → General → Common Language Runtime Support

The only coding you need to do in this project is to implement the factory function that will construct your `IDriver` object. This is done with the `Simba::CLDSI::LoadDriver` function:

```
Simba::DotNetDSI::IDriver^ Simba::CLDSI::LoadDriver()
{
    return gnew CustomerDotNetDSII::CustomerDSIIDriver();
}
```

For the purposes of examples, the output of this project will be referred to as `CustomerCLDSIDriver.dll`. This is the actual ODBC driver which will load the `CustomerDotNetDSII.dll` assembly.

6.2.4 Build as a SimbaServer (an EXE) for remote connections

In order to build a SimbaServer with or without Simba SQL Engine, in addition to the DotNet DSII project, you must create a CLIDSI project that provides the bridge between the native C++ SimbaServer libraries and your CustomerDotNetDSII.dll assembly. This is done by creating a project following the instructions above for C++ under Windows.

In addition to the library files listed in either 6.1.2 or 6.1.3 above, you must also include `CLIDSI_$(ConfigurationName).lib`. This library forms the bridge between unmanaged DSI classes to the managed DSI classes. The other configuration change you must make is to enable Common Language Runtime Support (/clr):

Configuration Properties → General → Common Language Runtime Support

The only coding you need to do in this project is to implement the factory function that will construct your `IDriver` object. This is done with the `Simba::CLIDSI::LoadDriver` function:

```
Simba::DotNetDSI::IDriver^ Simba::CLIDSI::LoadDriver()
{
    return gnew CustomerDotNetDSII::CustomerDSIIDriver();
}
```

For the purposes of examples, the output of this project will be referred to as `CustomerCLIDSIserver.exe`. This is the actual server executable which will load the `CustomerDotNetDSII.dll` assembly.

6.3 Java under Windows

Java drivers built using SimbaEngine SDK consist of two components: the native (C++) component and the Java DSII. For the native component, the following specifications refer to your Project Properties page in Microsoft Visual Studio 2008 or 2010. Please refer to the Microsoft MSDN documentation for a full listing of all compiler options. For the Java DSII, any special considerations are outlined.

6.3.1 Build as a JDBC Driver

The SimbaEngine JavaUltraLight sample driver demonstrates connections to SQL data sources using SimbaJDBC.

1. Ensure that the SimbaJDBC JAR file, located at `[INSTALL_DIRECTORY]\DataAccessComponents\Lib`, is in the classpath.
2. Create your driver DSII JAR file.

3. No additional libraries need to be linked.

6.3.2 Build as an ODBC Driver (a DLL) for local connections

The sample drivers discussed in the Build a Java ODBC Driver in 5 Days document are set up to build as Windows DLL's. To build your own driver as a Windows DLL, all the basic settings for C++ drivers from 6.1.1 apply to the native (C++) component with a few additions:

1. Include the additional directory for the JVM library that is under `$(JAVA_HOME)\lib`:

Configuration Properties → Linker → General → Additional Library Directories

Note that JAVA_HOME is an environment variable that should refer to the 32-bit Java installation directory when building the 32-bit ODBC driver or the 64-bit Java installation directory when building the 64-bit ODBC driver.

2. Link against `SimbaJNIDSI_$(ConfigurationName).lib` and `jvm.lib`:

Configuration Properties → Linker → Input → Additional Dependencies

ConfigurationName is one of these four values: `Debug`, `Debug_MTDLL`, `Release` or `Release_MTDLL`. See section 6.1.5, "Run-time library options" above for related settings for each option.

3. Include the general and Java include paths:

Configuration Properties → C/C++ → General → Additional Include Directories:

1. `$(SIMBAENGINE_DIR)\Include\JNIDSI`
2. `$(JAVA_HOME)\include`
3. `$(JAVA_HOME)\include\win32`

For the Java DSII, all you need to do is to ensure that the `SimbaJavaDSI.jar` is included in your build process. In the sample driver, the ANT build script packages the pre-compiled files with those of the DSII.

6.3.3 Build as a SimbaServer (an EXE) for remote connections

To build a driver as a stand-alone `SimbaServer` executable, the key settings for the native (C++) component are the same as for C++ drivers as described in 6.1.2. For the Java DSII, all you need to do is to ensure that the `SimbaJavaDSI.jar` is included in your build process. In the sample driver, the ANT build script packages the pre-compiled files with those of the DSII.

6.3.4 Building with Simba SQLEngine

The `SimbaEngine JavaQuickstart` sample driver demonstrates connections to non-SQL data sources using `Simba SQLEngine`. The settings for the native (C++) component are the same as for the C++ drivers as described in 6.1.3. For the Java DSII, all you need to do is to ensure that the `SimbaJavaDSI.jar` is included in your build process. In the sample driver, the ANT build script packages the pre-compiled files with those of the DSII.

6.3.5 Building without Simba SQLEngine

The SimbaEngine JavaUltraLight sample driver demonstrates connections to SQL data sources without using Simba SQLEngine.

1. All settings apply from either 6.3.2 or 6.3.3 above, depending on ODBC Driver or SimbaServer configuration.
2. No additional libraries need to be linked.
3. No additional paths need to be included.

6.4 Data Source Names and Driver entries in the Registry

The basic configuration of Data Source Names and Drivers in the Windows Registry is covered by the Build a Driver in 5 Days documents (each of the C++, C# and Java versions).

Adding Entries to `odbc.ini`

You can create your own parameters for use by the DSII layer. This section will describe where they are located on a 32-bit Windows machine.

IMPORTANT: The information in this section only applies if you are using 32-Bit Windows. If you are using 64-bit Windows (with either 32-bit or 64-bit applications), the file paths must be configured appropriately. Please see Appendix B: Windows Registry 32-Bit vs. 64-Bit in the Build a Driver in 5 Days documents (each of the C++, C# and Java versions) for information on the differences.

Table 2 below lists `odbc.ini` entries under the `HKEY_CURRENT_USER` key. Items in italics represent place holders for actual or string values. Note that the syntax shown in this listing—square brackets around section headings and equal signs separating parameters and values—is for example purposes only. Do not include the square brackets and equal signs in your string values.

Parameter	Description
[ODBC Data Sources]	(required) This section lists the data sources that the ODBC Driver Manager (and thus, ODBC applications) can access.
<i>DataSourceName=<The name of the driver as it appears under ODBCINST.INI registry key></i>	(required) A data source.
[DataSourceName]	(required) There must be a section for each data source named in the [ODBC Data Sources] section above.
<i>Driver=<path to driver>/driver.dll</i>	(required) This tells the ODBC Driver Manager the full name of the driver to load. This is the DLL, which you create by linking the SimbaEngine static libraries included in the SimbaEngine SDK

	with your DSII layer. The DLL name is what you named your driver: e.g. driver.dll.
Description=<data source description>	(optional) Description of the data source.
UID=<UserID>	(optional, data-source specific) User ID.
PWD=<Password>	(optional, data-source specific) Password.

Figure 15: Table of odbc.ini Parameters

6.5 C++ under Linux/Unix/MacOSX

This section outlines the options that are present in our sample Makefiles as well as the settings needed in your Makefile in order to compile your driver. See the Quickstart makefiles and any .mak files that it references for further information. Our sample makefiles for Quickstart include a reference to Platform.mak and other make files in the SDK detect the platform and choose the appropriate settings.

6.5.1 Sample Makefile Options

Our sample Makefiles specify a number of options that you may use to build the sample drivers. The following options are available for use (singly or in combination), depending on the platform.

Option	Description
ARCH	This option allows you to specify the bitness of the driver you wish to build. For example, if you use this option when you are building on a 64-bit platform, this will allow you to build a 32-bit driver.
BUILDSEVER	Use this option if you wish to build as a SimbaServer executable.
USE_GCC	This option specifies the use of the GCC compiler instead of the native compiler for building. It is currently required when using the sample makefiles on the AIX and Solaris platforms.

Figure 16: Description of sample makefile options

Platform	ARCH			USE_GCC	BUILDSEVER
	32-bit	64-bit	Itanium (64-bit)		
AIX	powerpc	powerpc64	-	1	exe
Darwin	x86	X8664	-	-	-
HP-UX	-	-	ia64	-	exe
Linux	x86	x8664	ia64	-	exe

Solaris Sparc	sparc	sparc64	-	1	exe
Solaris x86	x86	x8664	-	1	exe

Figure 17: Sample makefile option values by platform. '-' Indicates that the option is not available.

To use these options, you can specify them on the commandline. For example, to build the release configuration for a 32-bit Quickstart server on a 64-bit Linux, you can specify:

```
make -f Quickstart.mak release ARCH=x86 BUILDSERVER=exe
```

If you do not specify any options, a standalone driver will be built using the default native compiler.

6.5.1.1 Build as an ODBC Driver (a Shared Object) for local connections

To build a driver as a Shared Object, the settings needed are:

1. Set compiler and linker to build a shared object (option name depends on compiler).
2. Include the following Simba libraries when linking:

1. libSimbaODBC_<TARGET>.a
2. libSimbaDSI_<TARGET>.a
3. libSimbaSupport_<TARGET>.a

<TARGET> is one of these four values: debug, debug_unixODBC, release or release_unixODBC

See section 6.5.5, "Build configurations" below for an explanation and notes on each option.

3. Set the symbol exports file when linking:

Filename and options depend on compiler. See Quickstart for examples. For example, in Quickstart under Linux, add this to the link command: `-Wl,--version-script=exports_Linux.map`

4. Include the DSI and Support include paths:

1. \$(SIMBAENGINE_DIR)\Include\DSI
2. \$(SIMBAENGINE_DIR)\Include\DSI\Client
3. \$(SIMBAENGINE_DIR)\Include\Support
4. \$(SIMBAENGINE_DIR)\Include\Support\Exceptions
5. \$(SIMBAENGINE_DIR)\Include\Support\TypedDataWrapper

Sample makefiles are included in Appendix K: Sample Unix Makefiles to build as a Shared Object on page 132.

6.5.2 Build as a SimbaServer (an Executable) for remote connections

To build a driver as a stand-alone SimbaServer executable, the key settings needed are:

1. Set compiler and linker to build an executable (option name depends on compiler).
2. Include the following Simba libraries when linking:

1. libSimbaServer_<TARGET>.a
2. libSimbaCommunications_<TARGET>.a
3. libSimbaMessages_<TARGET>.a
4. libSimbaDSI_<TARGET>.a
5. libSimbaSupport_<TARGET>.a

<TARGET> is one of these four values: debug, debug_unixODBC, release or release_unixODBC

See section 6.5.5, "Build configurations" below for an explanation and notes on each option.

3. Include the DSI and Support include paths:

1. \$(SIMBAENGINE_DIR)\Include\DSI
2. \$(SIMBAENGINE_DIR)\Include\DSI\Client
3. \$(SIMBAENGINE_DIR)\Include\Support
4. \$(SIMBAENGINE_DIR)\Include\Support\Exceptions
5. \$(SIMBAENGINE_DIR)\Include\Support\TypedDataWrapper

Sample makefiles are included in Appendix M: Sample Unix Makefiles to build as a SimbaServer on page 144.

6.5.3 Building with Simba SQLEngine

The SimbaEngine Quickstart sample driver demonstrates connections to a non-SQL data source using Simba SQLEngine.

1. All settings apply from either 6.5.1 or 6.5.2 above, depending on ODBC Driver or SimbaServer configuration.
2. Include the following additional Simba libraries when linking:

1. libDSIExt_<TARGET>.a
2. libCore_<TARGET>.a
3. libExecutor_<TARGET>.a
4. libParser_<TARGET>.a
5. libAETProcessor_<TARGET>.a

<TARGET> is one of these four values: debug, debug_unixODBC, release or release_unixODBC

See section 6.5.5, "Build configurations" below for an explanation and notes on each option.

3. Include the SQLEngine include paths:

1. \$(SIMBAENGINE_DIR)\Include\SQLEngine
2. \$(SIMBAENGINE_DIR)\Include\SQLEngine\AETree
3. \$(SIMBAENGINE_DIR)\Include\SQLEngine\DSIExt

6.5.4 Building without Simba SQLEngine

The SimbaEngine Ultralight sample driver demonstrates a connection to a data source that provides SQL processing.

1. All settings apply from either 6.5.1 or 6.5.2 above, depending on ODBC Driver or SimbaServer configuration.
2. No additional libraries need to be linked.
3. No additional paths need to be included.

6.5.5 Build configurations

There are four build configurations (`debug`, `debug_unixODBC`, `release` and `release_unixODBC`) for each Simba library.

debug

Debug version of libraries used to build most drivers and server.

debug_unixODBC

Debug version of libraries used to build drivers compatible with unixODBC 2.2.12 and earlier. Should not be used for unixODBC 2.2.13 or newer, other driver managers, or building a server.

See Appendix L: Building a driver for UnixODBC 2.2.12 and earlier on page 143 for example changes to sample Makefiles.

release

Release version of libraries used to build most drivers and server.

Library names do not contain `_release`. ie. `libDSI_<TARGET>.a` becomes `libDSI.a`

release_unixODBC

Release version of libraries used to build drivers compatible with unixODBC 2.2.12 and earlier. Should not be used for unixODBC 2.2.13 or newer, other driver managers, or building a server.

Library names do not contain `_release`. ie. `libDSI_<TARGET>.a` becomes `libDSI_unixODBC.a`

See Appendix L: Building a driver for UnixODBC 2.2.12 and earlier on page 143 for example changes to sample Makefiles.

6.6 Java under Linux/Unix/MacOSX

Java drivers consist of two components: the native (C++) component and the Java DSII. For the native component, this section outlines the settings needed in your Makefile in order to compile your driver. For information on the options that are present in our sample Makefiles, please refer to 6.5.1 Sample Makefile Options. See the JavaQuickstart makefiles and any `.mak`

files that it references for further information. Our sample makefiles for JavaQuickstart include a reference to Platform.mak and other make files in the SDK detect the platform and choose the appropriate settings. For the Java DSII, any special considerations are outlined.

6.6.1 Build as a JDBC Driver

The SimbaEngine JavaUltraLight sample driver demonstrates connections to SQL data sources using SimbaJDBC.

4. Ensure that the SimbaJDBC JAR file, located at `[INSTALL_DIRECTORY]\DataAccessComponents\Lib`, is in the classpath.
5. Create your driver DSII JAR file.
6. No additional libraries need to be linked.

6.6.2 Build as an ODBC Driver (a Shared Object) for local connections

To build a driver as a Shared Object, all the basic settings for C++ drivers from 6.5.1 apply to the native (C++) component with a few additions:

1. Include the following libraries when linking:

1. `libSimbaJNIDSI_<TARGET>.a`

<TARGET> is one of these four values: `debug`, `debug_unixODBC`, `release` or `release_unixODBC`

See section 6.5.5, "Build configurations" above for an explanation and notes on each option.

2. Link in the JVM library:

Location depends on bitness and the Java installation. See JavaQuickstart for examples. For example, for a 64-bit JavaQuickstart under Linux, add this to the link command:

```
-L$(JAVA_HOME)/jre/lib/amd64/server -ljvml
```

Note that JAVA_HOME is an environment variable that should refer to the appropriate Java installation directory. For a Java Development Kit (JDK), the location of the JVM on 32-bit Unix is usually in `<Java Home>/jre/lib/<architecture>/client` (example: `<architecture>` on Linux is `i386`), while on 64-bit Unix it is usually in `<Java Home>/jre/lib/<architecture>/server` (example: `<architecture>` on Linux is `amd64`).

3. Include the JNIDSI and Java include paths:

1. `$(JAVA_HOME)/include`

2. `$(SIMBAENGINE_DIR)/Include/JNIDSI`

For the Java DSII, all you need to do is to ensure that the `SimbaJavaDSI.jar` is included in your build process. In the sample driver, the ANT build script packages the pre-compiled files with those of the DSII.

6.6.3 Build as a SimbaServer (an Executable) for remote connections

To build a driver as a stand-alone SimbaServer executable, the key settings for the native (C++) component are the same as for C++ drivers as described in 6.5.2. For the Java DSII, all you need to do is to ensure that the SimbaJavaDSI.jar is included in your build process. In the sample driver, the ANT build script packages the pre-compiled files with those of the DSII.

6.6.4 Building with Simba SQLEngine

The SimbaEngine JavaQuickstart sample driver demonstrates connections to non-SQL data sources using Simba SQLEngine. The settings for the native (C++) component are the same as for the C++ drivers as described in 6.5.3. For the Java DSII, all you need to do is to ensure that the SimbaJavaDSI.jar is included in your build process. In the sample driver, the ANT build script packages the pre-compiled files with those of the DSII.

6.6.5 Building without Simba SQLEngine

The SimbaEngine JavaUltraLight sample driver demonstrates connections to SQL data sources without using Simba SQLEngine.

1. All settings apply from either 6.6.1 or 6.6.3 above, depending on ODBC Driver or SimbaServer configuration.
2. No additional libraries need to be linked.
3. No additional paths need to be included.

6.7 Access to Data Sources under Linux/Unix/MacOSX

6.7.1 Linux/Unix/MacOSX Driver Managers

Linux and Unix installations do not come with a Driver Manager as part of the operating system as Windows does. You will need to install your own driver manager before you can compile and test your driver under Linux or Unix. Two popular choices are:

- iODBC, available here: www.iodbc.org.
- UnixODBC, available here: www.unixodbc.org.

6.7.2 Configuring Data Sources under Linux/Unix/MacOSX

On Linux and Unix the equivalent configuration is handled via `ODBC.INI` and `ODBCINST.INI` files located in your `$HOME` directory. Samples of these files are provided from which you can copy driver and DSN information to modify your existing `ODBC.INI` and `ODBCINST.INI` files.

If these files do not already exist on your system, you can copy the samples to your *\$HOME* directory.

The ODBC Data Source and ODBC Drivers are specified in two .ini files: `odbc.ini` and `odbcinst.ini`. The `odbc.ini` file defines the DSNs and the `odbcinst.ini` file defines the drivers. The `odbc.ini` and `odbcinst.ini` files are located using the Driver Manger's library. Typically the search order is:

ODBC.INI

1. If an environment variable is defined (such as `ODBCSYSINI`, depending on the Driver Manager), then the value of the variable will be the first directory searched in for the `odbc.ini` file.
2. The next directory that will be searched is `~/` (i.e. `$HOME`)

Note: the file must have a preceding 'dot' i.e. `~/odbc.ini`

3. Finally, the system-wide default `/etc/odbc.ini` will be used. Note the lack of a preceding 'dot'.

ODBCINST.INI

1. If an environment variable is defined (such as `ODBCSYSINI`, depending on the Driver Manager), then the value of variable will be the first directory searched in for the `odbcinst.ini` file.
2. The next directory that will be searched is `~/` (i.e. `$HOME`)

Note: the file must have a preceding 'dot' i.e. `~/odbcinst.ini`

3. Finally, the system-wide default `/etc/odbcinst.ini` will be used.

Note the lack of a preceding 'dot'.

SIMBA.INI

You will also need to create a configuration file that is specific for each driver in your home directory. For example, the Quickstart driver would need the configuration file `.simba.quickstart.ini`. Note the preceding 'dot'. This ini file specifies certain driver-wide settings.

The search order for this file is as follows:

1. If the `SIMBAINI` environment variable is defined, then the value of `SIMBAINI` is used to locate the file. `SIMBAINI` must contain the full path including the filename.

2. Otherwise, if the SIMBAINI environment variable is not defined, the current working directory of the application will be searched for simba.ini.

Note the lack of a preceding 'dot'.

3. The next directory that will be searched is ~/ (i.e. \$HOME) for .simba.ini.
4. Finally, the system wide default /etc/simba.ini will be used.

Note the lack of a preceding 'dot'.

The steps to create this file are:

1. Create a new section called [Driver].
2. Add a key to this section, called **DriverManagerEncoding**. The DriverManagerEncoding is the encoding used for Unicode strings by the Driver Manager. Please see the matrix in Frequently Asked Questions on page 110 for the appropriate encoding (UTF-16 or UTF-32) to use for your platform and environment.
3. Add a key called **ErrorMessagePath**. The value of this key should be:

```
<install dir>/ErrorMessages/
```
4. Add a key called **ODBCInstLib**. The ODBCInst library is a part of the Driver Manager but is used by the Driver to read values from the odbc.ini file. The value of this key is the absolute path of the ODBCInst library and depends on which Driver Manager you are using. For example, for the iODBC Driver Manager this would be `<driver manager dir>/lib/libiodbcinst.so` (notice the 'i' after the lib) and for unixODBC this would be `<driver manager dir>/lib/libodbcinst.so`
5. Add a key called **LogLevel**. The value of this key can be 0 to turn off logging, and a value of up to 6 to turn on logging. This logging feature, when turned on, will create a log for each connection that your driver establishes to your data source. Every event that occurs in the Data Store Interface layer will be written to this log. This feature is very useful while developing and debugging your driver. It should, however, be turned off once your driver is complete for optimal performance.

Listed below are sample odbc.ini, odbcinst.ini and simba.ini files. You need only to replace the DSN, driver, and paths with the appropriate values and to create the files in the appropriate directories as per the search procedure discussed above.

Example ODBC.INI file:

```
[ODBC Data Sources]
myDSN=myDriver

[myDSN]
Driver=<full path to myDriver.so>
Description=my DSN
```

```
[ODBC]
Trace=1
TraceFile=<dir to trace file>/ODBCTrace-$U-$P-$T.log
```

Example ODBCINST.INI file:

```
[ODBC Drivers]
myDriver=Installed

[myDriver]
Driver=<full path to myDriver.so>
```

Example SIMBA.INI file:

```
[Driver]
DriverManagerEncoding=UTF-32
ErrorMessagesPath=<install dir>/ErrorMessages/
ODBCInstLib=<driver manager dir>/lib/libiodbcinst.so
```

7 Testing your DSN

The Build a Driver in 5 Days documents (each of the C++, C# and Java versions) walk you through testing your driver, focusing primarily on the Windows platform. This section goes into more depth on the subject and the variations for other platforms.

7.1 Testing under Windows

7.1.1 Microsoft Access

Running your driver against Microsoft Access is a good test to prove basic functionality. MS Access utilizes much of the ODBC API and shows many of the API's quirks. Test your driver under MS Access by loading your data as linked tables. This method uses the greatest breadth of the ODBC API.

A quick test run

1. Open Access and create a new "Blank Database".
2. Under "External Tab", select "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 "Select Data Source" dialog, select "Machine Data Source" table and choose your DSN. Click OK.
5. In "Link Tables" dialog, select the tables you want to link to. Click OK.
6. In "Select Unique Record Identifier" dialog, click OK without choosing any specific field.
7. In "All Table" panel, right click on a table name and click "Open". You should see the data from the table in Access.

7.1.2 Microsoft Excel

You can also test your data source with Microsoft excel by importing data using the "Data connection wizard".

A quick test run

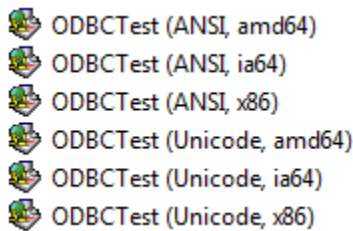
1. Open Excel and create a new "Blank Workbook".
2. Under the "Data" tab, select "From Other Sources" -> "From Data Connection Wizard"
3. In "Data Connection Wizard" dialog, select "ODBC DSN" and click "Next".
4. Select your DSN in "ODBC data sources" box, and click "Next".
5. Select a table from the "Table" box, and click "Next" and then "Finish"

6. In “Import Data” dialog, select “Table” for how you want to view the data in the workbook and click “OK”. You should see the data from the table in Excel.

7.1.3 ODBCTest

ODBCTest is a test application provided by Microsoft as part of the Microsoft Data Access Components (MDAC) SDK and the Platform SDK (see section 9.2, “What is MDAC?” for more information). This application allows you to execute arbitrary SQL queries simply, or call any method in the ODBC API directly. Thus you can state-fully test your driver step-by-step and view the results of each ODBC API call live.

The 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; ensure that you select the appropriate one for the type of testing needed:



Take special note: On 64-bit Windows, in addition to running the right version of the application, you must also ensure that your DSNs are configured for the correct version, see Appendix A: Windows 32-Bit vs. 64-Bit in the Build a Driver in 5 Days documents (each of the C++, C# and Java versions) for details.

Running your driver 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 trace down problems precisely.

A quick test run

1. Start the ODBCTest application (remember as noted above to run the correct version for ANSI or Unicode and 32-bit or 64-bit).
2. To configure the application to use ODBC3.52 menus, select Options from the Tools menu. Select the ‘ODBC Menu Version’ tab and select ‘ODBC 3.x’.
3. Form a connection to your driver using the appropriate DSN you already configured (see Data Source Names and Driver entries in the Registry Section 6.4, “Data Source Names and Driver entries in the Registry” on page 85 for more information).
 - a. From the “Connect” menu, select “Full Connect”. This option does all the work of allocating environment and connection handles, and opening the connection.
 - b. Locate your DSN in the data source list.

4. Hit OK. You should have a new window appear with the message “Successfully connected to DSN ‘<your driver name>” at the bottom.

Help! I got an error message “SQLDriverConnect returned: SQL_ERROR=-1”.

This Error usually occurs because the Windows Driver Manager cannot find or load the requested driver’s DLL. Here are some things to check:

- 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 driver 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?
5. The top half of the window that just appeared allows you to enter SQL queries to be executed. The bottom half displays the results.
 6. Enter a simple SQL query now
 7. From the “Statement” menu select “SQLExecDirect” and hit OK on the resulting dialog.
 8. From the “Results” menu select “GetDataAll”
 9. Review the results
 10. From the “Catalog” menu select “SQLTables” and hit OK on the resulting dialog. Note: SQL Catalog functions work only if you have implemented the appropriate MetadataSources.
 11. From the “Results” menu select “GetDataAll”
 12. Review the results
 13. The full ODBC API is exposed through the menus. Have fun!

7.2 Testing under Linux/Unix/MacOSX

7.2.1 iODBCTest and iODBCTestW

iodbctest and iodbctestw are utilities included with the iODBC driver manager installation. You can use one or the other of these utilities to establish a test connection with your driver and your DSN. Use iodbctest to test how your driver works with an ANSI application. Use iodbctestw to test how your driver works with a Unicode application.

Refer to the www.iodbc.org website for further details about how to use this utility.

1. Run iodbctest (or iodbctestw).

```
./iodbctest
```

or

```
./iodbctestw
```

Take special note: There are 32-bit and 64-bit installations of the iODBC driver manager available. If you have only one or the other 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 take care that you are running the version from the correct installation directory.

2. The program will ask you to enter an 'ODBC connect string'. You can type ? if you do not remember the name of your DSN. Your ODBC connect string will have the following format:

```
DSN=<your_DSN_name>;UID=<user_id> (if applicable);PWD=<your password> (if applicable)
```

3. If you have successfully connected, you will see a `SQL>` prompt.
4. Test out some simple `SELECT` queries to see if your data is being retrieved properly from your data source.

7.2.2 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 driver and your DSN.

Refer to the www.unixodbc.org website for further details about how to use this utility.

1. Run `iSql`:

```
./isql <DSN> <UID (if applicable)> <PWD (if applicable)> <options (if applicable)>
```

2. If you have successfully connected, you will see `SQL>` prompt.
3. Test out some simple `SELECT` queries to see if your data is being retrieved properly from your data source.

7.2.3 Logging

The SDK includes comprehensive logging functionality for both developing your driver, and helping Simba diagnose problems in DSIs. The logging is split so that there are 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 DSI. 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.

Java DSI Specific Features

To ease implementation of logging in DSIs using the Java DSI, the SDK includes a helper class called `LogUtilities`. 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.

DotNet DSI Specific Features

To ease implementation of logging in DSIs using the DotNet DSI, the SDK includes a helper class called `LogUtilities`. 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 DSI DLL is loaded, the DSI `ILogger` will be used.

8 Packaging Your Driver

8.1 Preparing your driver to ship to end users

If you build your driver following the local model, you will need to install the driver on end user machines in order for the desktop applications to be able to use it.

Note: If you build your driver following the remote model with SimbaServer, then you simply ship the already built SimbaClients to your end users—see the [SimbaClientServer User Guide](#) for more details.

This section outlines the files you will need to include in the package you ship.

8.1.1 C++ Packaging for Windows

A C++ based driver built for Windows needs to include the following:

1. The driver DLL and any new dependency referenced DLLs you add.
2. ICU DLLs (see Appendix I: Build Platform on page 131 to determine the value to use for <PLATFORM> below):

- For 32-bit drivers, include these files from:

```
[INSTALL_DIRECTORY]\DataAccessComponents\ThirdParty\icu\<PLATFORM>\lib\
```

- o simbaicudt38_32.dll, simbaicuin38_32.dll, simbaicuuc38_32.dll

- For 64-bit drivers, include these files from:

```
[INSTALL_DIRECTORY]\DataAccessComponents\ThirdParty\icu\<PLATFORM>\lib\
```

- o simbaicudt38_64.dll, simbaicuin38_64.dll, simbaicuuc38_64.dll

3. Error message file, `ErrorMessages.xml`, located at:

```
[INSTALL_DIRECTORY]\DataAccessComponents\ErrorMessages\
```
4. Add the following registry entries under the `HKLM\Software\Simba\Driver` key (or for a 32-bit driver on a 64-bit system, under the `HKLM\Software\Wow6432Node\Simba\Driver` key):
 - `DriverManagerEncoding = UTF-16`
 - `ErrorMessagesPath = <Path to the directory where error message file is located>`
 - (OPTIONAL) `LogLevel = 0`
 - (OPTIONAL) `LogPath = <Path to directory to store the log files>`

5. For a 32-bit driver, add the following registry entry under the HKLM\Software\ODBC\ODBCINST.INI\ODBC Drivers key (or for a 32-bit driver on a 64-bit system, under the HKLM\Software\Wow6432Node\ODBC\ODBCINST.INI\ODBC Drivers key):
 - `<DRIVER_NAME>=Installed`
6. For a 32-bit driver, add the following registry entries under the HKLM\Software\ODBC\ODBCINST.INI\<DRIVER_NAME> key (or for a 32-bit driver on a 64-bit system, under the HKLM\Software\Wow6432Node\ODBC\ODBCINST.INI\<DRIVER_NAME> key):
 - `Driver=<Full path to the 32-bit driver DLL>`
 - `Description=<Brief description of your driver>`
 - `Setup=<Full path to the 32-bit driver configuration dialog DLL>`
7. For a 64-bit driver, add the following registry entry under the HKLM\Software\ODBC\ODBCINST.INI\ODBC Driver key for your driver to be recognized by the 64-bit ODBC Admin:
 - `<DRIVER_NAME>=Installed`
8. For a 64-bit driver, add the following registry entries under the HKLM\Software\ODBC\ODBCINST.INI\<DRIVER_NAME>:
 - `Driver=<Full path to the 64-bit driver DLL>`
 - `Description=<Brief description of your driver>`
 - `Setup=<Full path to the 64-bit driver configuration dialog DLL>`

8.1.2 C++ Packaging for Linux/Unix/MacOSX

A C++ based driver built for Linux/Unix/MacOSX needs to include the following:

1. The driver shared object and any new dependency referenced shared objects you add.
2. ICU shared objects (see Appendix 1: Build Platform on page 131 to determine the value to use for <PLATFORM> below):
 - Include all files from:
`[INSTALL_DIRECTORY]/DataAccessComponents/ThirdParty/icu/<PLATFORM>/lib/`
3. Error message file, `ErrorMessages.xml`, located at:
`[INSTALL_DIRECTORY]/DataAccessComponents/ErrorMessage/`
4. Add the following entries to `.simba.<oem name>.ini`:
 - `DriverManagerEncoding=UTF-16` (or UTF-32, depending on the driver manager being used)

- ErrorMessagePath=<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 driver shared library>
 - Description=<Brief description of your driver>

8.1.3 C# Packaging (Windows only)

With Simba.NET

A C# based driver requires:

6. The C# driver DLL.
7. The SDK's C# DLLs, `Simba.DotNetDSI.dll` and `Simba.ADO.Net.dll` from `[INSTALL_DIRECTORY]\Bin\Win`
8. Three of the DLLs have to be installed to the Global Assembly Cache (GAC) on the target machine using the `gacutil.exe` utility:
 - `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)

With CLI DSI and/or SQLEngine

In addition to the C++ Packaging for Windows requirements described in section 8.1.1 above, a C# based driver also requires:

1. The driver's CLIDSI DLL in addition to the C# driver DLL.
2. The SDK's C# DLLs, `Simba.DotNetDSI.dll` and `Simba.DotNetDSIExt.dll` from `[INSTALL_DIRECTORY]\Bin\Win`
3. The `Driver` entry in the registry should be the full path to the C++ CLIDSI DLL, and not the C# DLL.
4. The CLIDSI error message file, `CLIDSIMessages.xml`, located at:
`[INSTALL_DIRECTORY]\DataAccessComponents\ErrorMessages\`
5. Three of the DLLs have to be installed to the Global Assembly Cache (GAC) on the target machine using the `gacutil.exe` utility:
 - `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)

8.1.4 Java Packaging for Windows

With SimbaJDBC

A Java based driver requires:

1. The SimbaJDBC JAR file located at
`[INSTALL_DIRECTORY]\DataAccessComponents\Lib`
2. The Java driver's JAR file.

With JNI DSI and/or SQLEngine

In addition to the C++ Packaging for Windows requirements described in section 8.1.1 above, a Java based driver also requires:

1. The driver's JNIDSI DLL in addition to the Java driver JAR file.
2. The `Driver` entry in the registry should be the full path to the C++ JNIDSI DLL, and not the driver's Java JAR file.
3. The JNIDSI error message file, `JNIDSIErrorMessages.xml`, located at:
[INSTALL_DIRECTORY]\DataAccessComponents\ErrorMessages\
4. Add the following registry entry under the `HKLM\Software\Simba\Driver` key (or for a 32-bit driver on a 64-bit system, under the `HKLM\Software\Wow6432Node\Simba\Driver` key):

- `JNIConfig=<Java Virtual Machine (JVM) Configuration options>`

If multiple options, they are separated by a "|" character. For example:

```
-Djava.class.path=<full path>\JavaQuickstart.jar|-Xdebug
```

5. Add/modify the following environment variables:

- `CLASSPATH=<Full path of the driver's JAR file>`

Only if `-Djava.class.path` is not specified in the `JNIConfig`. For example:
`<full path>\JavaQuickstart.jar`

- `PATH=<Location of the Java executable as well as the 64-bit or 32-bit Java Virtual machine (Depending on the bitness of JNIDSI driver)>`

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`.

8.1.5 Java Packaging for Linux/Unix/MacOSX

With SimbaJDBC

A Java based driver requires:

1. The SimbaJDBC JAR file located at
[INSTALL_DIRECTORY]\DataAccessComponents\Lib
2. The Java driver's JAR file.

With JNI DSI and/or SQLEngine

In addition to the C++ Packaging for Linux/Unix/MacOSX requirements described in section 8.1.2 above, a Java based driver also requires:

1. The driver's JNIDSI library in addition to the Java driver JAR file.
2. The `Driver` entry in the registry should be the full path to the C++ JNIDSI DLL, and not the driver's Java JAR file.
3. The JNIDSI error message file, `JNIDSIErrorMessages.xml`, located at:
[INSTALL_DIRECTORY]/DataAccessComponents/ErrorMessages/
4. Add the following entry to `.simba.ini`:
 - `JNIConfig=<Java Virtual Machine (JVM) Configuration options>`

If multiple options, they are separated by a "|" character. For example:

```
-Djava.class.path=<full path>/JavaQuickstart.jar|-Xdebug
```

5. Add/modify the following environment variables:

- `CLASSPATH=<Full path of the driver's JAR file>`

Only if `-Djava.class.path` is not specified in the `JNIConfig`. For example:
`<full path>/JavaQuickstart.jar`

- `LD_LIBRARY_PATH=<Addition of the 64-bit or 32-bit Java Virtual machine (Depending on the bitness of JNIDSI driver)>`

For a Java Runtime Environment (JRE), the location of the JVM on 32-bit Unix is usually in `<JRE Path>/lib/<architecture>/client` (example: `<architecture>` on Linux is `i386`), while on 64-bit Unix it is usually in `<JRE path>/lib/<architecture>/server` (example: `<architecture>` on Linux is `amd64`).

8.2 Create a Custom DSN Configuration Application

One of the best things you can do as a next step is to create a DSN Configuration Application. This is the dialog that is presented when you click the Add, Remove, or Configure buttons in the ODBC Data Source Administrator. End-users of your ODBC-enabled application and driver can use your custom DSN configuration application to simplify set-up. This is because they will not need to manually modify or create registry entries (Windows) or .ini files (Unix).

To do this, you will need to implement the Driver Setup DLL API. Specifically, this means implementing and exporting the `ConfigDSN` function:

```
BOOL INSTAPI ConfigDSN(
    HWND in_parentWindow,
```

```
WORD in_requestType,
LPCSTR in_driverDescription,
LPCSTR in_attributes)
```

You may implement this in the driver DLL or as a separate DLL. Note that when building the config as a separate DLL, a common convention is to change the file extension to CNF. Both Quickstart and Codebase provide examples of implementing ConfigDSN and exporting it from within the driver DLL.

To get your setup function recognized by the ODBC Data Source Administrator, you must add the Setup key to your driver's entry in ODBCINST.INI in the registry. See above in the C++ Packaging for Windows section for an example.

Refer to the Setup DLL API Reference and the Installer DLL API Reference Function in the Microsoft ODBC Programmer's Reference.

8.3 How To Build Your Driver As An OEM Solution

8.3.1 ODBC

The default Simba branding can be removed from a driver and replaced with an OEM branding with the following changes:

1. Change name for driver configuration files/registry entries:
 - a. Call `SimbaSettingReader::SetConfigurationBranding(<OEM NAME>)` from the `Simba::DSI::DSIDriverFactory()` function.
 - i. On Windows, this causes the configuration to be read from the registry key: `HKLM/Software/<OEM NAME>/Driver/`
 - ii. On Unix/Linux, this causes the configuration to be read from the `<OEM NAME>` file. The search order for this file is:
 1. Current Working Directory of the client application.
 2. Home directory with a `.'` (dot) prefix on the filename.
 3. `/etc/`
 - b. For a Server DSN, also call `Simba::Server::SetServerConfigurationBranding(<OEM NAME>);`
 - i. On Windows, this causes the server configuration to be read from the registry key: `HKLM/Software/<OEM NAME>/Server/`
 - ii. On Unix/Linux, the file search order is the same as above. The filename may be the same or different from the driver configuration.

- c. For Unix/Linux, it may be desirable to place the <OEM NAME> file in a different location. In this case, use the SIMBAINI environment variable to change the location of the search directory.
 - i. To rebrand the SIMBAINI environment variable, call `SimbaSettingReader::SetUnixConfigEnvVariable(<ENV NAME>)` from the `Simba::DSI::DSIDriverFactory()` function.
 - ii. To export this variable, from the command line export the variable to the correct path. For example:


```
export SIMBAINI=/path/to/<ENV NAME>.ini
```
2. Call `SetVendorName(<OEM NAME>)` on the `DSIMessageSource` object that is created by the `DSIDriver`. This may be done from inside the constructor of a derived class, or after instantiating it the message source.
 - a. This changes the name that shows up in error messages to be of the form: `[<OEM NAME>][Component Name] Error message...`

8.3.2 JDBC

The default Simba branding can be removed from a driver and replaced with an OEM branding with the following changes:

1. Change name for logging entries:
 - a. When registering a messages file with `DSIMessageSource::registerMessages()`, use <OEM NAME> as the third parameter to the function.
 - i. This changes the name that shows up in error messages to be of the form: `[<OEM NAME>][Component Name] Error message...`
2. To remove the simba package from exception stack traces, Simba can provide a utility to partially obfuscate the provided SimbaJDBC JAR file. This allows the simba package to be replaced with <OEM NAME>.
 - a. This is called the `SimbaPackageRenamer.jar` file.
 - b. From the command line, navigate to the utility JAR file and execute the following:


```
java -jar SimbaPackageRenamer.jar <JAR_FILE> <OEM NAME>
```

 where <JAR_FILE> is the fully qualified path to the `SimbaJDBC.jar` file.
3. You may optionally choose to subclass some core JDBC classes to provide branding or extended functionality:
 - a. The classes available for subclassing are `SConnection`, `SStatement`, `SPreparedStatement`, `SCallableStatement`, and `SDatabaseMetaData`.

- b. Implement and override the virtual method `AbstractDriver::createJDBCObjectFactory()` and `AbstractDataSource::createJDBCObjectFactory()` to return a subclass of `JDBCObjectFactory`.
- c. Implement and override the virtual method in your subclass of `JDBCObjectFactory` that corresponds to the core class that you would like to subclass and extend so that it returns your custom subclass.

8.3.3 ADO.NET

The default Simba branding can be removed from a driver and replaced with an OEM branding with the following changes:

1. Change name for logging entries:
 - a. When registering a messages file with `DSIMessageSource::registerMessages()`, use `<OEM NAME>` as the third parameter to the function.
 - i. This changes the name that shows up in error messages to be of the form: `[<OEM NAME>][Component Name] Error message...`
2. Subclass core Simba.NET and ADO.NET classes:
 - a. There are a certain set of core ADO.NET classes which need to be subclassed by your .NET driver, and provide branding as well. These are: `SCommand`, `SCommandBuilder`, `SConnection`, `SConnectionStringBuilder`, `SDataAdapter`, `SFactory`, and `SParameter`.
 - b. You may optionally also subclass `SDataReader`.
 - c. See C# under Windows for more information.

9 Frequently Asked Questions

This section answers the most commonly asked questions about our product and technology.

9.1 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 driver; SIMBA.DLL, and standards-based data access was born. Using ODBC, you can access data stored in many common databases. A separate ODBC driver 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.

9.2 What is MDAC?

MDAC stands for Microsoft Data Access Components. Runtime components are shipped with the Windows operating system and contain interfaces for ODBC, OLEDB and ADO as well as the ODBC drivers 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. See this link for more information: <http://msdn.microsoft.com/en-us/library/ms692877%28v=vs.85%29.aspx>.

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

ODBC-enabled applications always access ODBC Drivers 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 Driver to use when establishing a connection. The Driver Manager will then load the appropriate ODBC Driver. Once the ODBC Driver is loaded, the Driver Manager will map all incoming requests to the appropriate functions exported by the ODBC Driver.

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 driver 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 Driver 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:

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

9.4 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>

9.5 What is SimbaODBC?

SimbaODBC is a component part of the SimbaEngine SDK for developing full-featured, optimized ODBC 3.52 drivers on top of any SQL-enabled data source. SimbaODBC provides extensibility for JDBC, OLE DB as well as ADO.NET connectivity. The Build a Driver in 5 Days documents will provide you with sufficient information to rapidly develop a read-only driver, which can then be extended to include additional functionality and optimizations. SimbaODBC simplifies exposing the query parsing, query execution and data retrieval facilities of your SQL-enabled data source.

9.6 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.

9.7 How do I build an ODBC driver using SimbaEngine?

You can build an ODBC Driver for a SQL-enabled data store using the SimbaODBC component by referring to the SimbaEngine User's Guide. To build an ODBC Driver for a non-SQL-enabled data store, please refer to the Build a Driver in 5 Days documents (there is one for each supported language, C++, Java and C#).

9.8 How can I obtain more information about SimbaEngine SDK?

Please visit our website at <http://www.simba.com> for additional resources, or contact us if you have any specific questions.

9.9 What SQL conformance level does SimbaEngine SDK support?

The SimbaEngine SDK supports the full core-level ODBC 3.52. It supports most of the Level 1 and Level 2 API.

Conformance Level	INTERFACES	Conformance Level	INTERFACES
Core	SQLAllocHandle	Core	SQLGetInfo
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
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

Figure 18: Table of ODBC 3.52 Interfaces Supported by SimbaODBC.

The ODBC version 3.52 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.

Appendix A: Platforms and System Requirements

This appendix details the minimum hardware and software requirements for installing and working with SimbaEngine SDK.

Hardware Requirements

For both Windows and Linux/Unix systems, the hardware requirements are the same:

- 4 GB of free disk space
- 1 GB RAM

Software Requirements

This table lists all of the supported platform combinations as of the time this document was published:

Platform	Version	Compiler	Bits
Windows	XP, Vista, 7, Server 2003, Server 2008	Visual Studio 2008, 2010	32, 64
Linux	CentOS/RHEL 4, 5; SLES 10, 11	GNU GCC	32, 64
Solaris (x86)	10, 11	Solaris C compiler version 5.8	32, 64
Solaris (x86)	10, 11	GNU GCC	32, 64
AIX	5.3, 6.1	IBM XL C version 8.0	32
AIX	5.3, 6.1	XL C/C++ for AIX V11.1	64
HP-UX (IA-64)	11.23 and above	aCC version A.03.70	64
Mac OS	10.6 and above	GNU GCC	32, 64

Java

If you are using Java, the following version information applies:

- JDK 1.5 or higher
- JRE 1.5 or higher
- JDBC 3.0

Appendix B: Errors and Exceptions

Throwing Exceptions

When your DSII detects an error condition, you will throw an exception called `ErrorException`. The basic signature for it is:

```
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 key arguments to this when defining your own errors are `in_componentId` and `in_msgKey`. The component Id is to be used to determine what 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. The `in_msgKey` argument is a string shortcut to indicate which message to load from the xml file or your own custom message source.

The `in_stateKey` argument is used to control which `SQLSTATE` code should be associated with the error returned by ODBC. The most common state to throw is `DIAG_GENERAL_ERROR`. A full list of available `DiagState` keys can be found in `DiagState.h`.

The sample drivers `Quickstart` and `Codebase` provide sample macros (defined in `Quickstart.h` and `Codebase.h`) which you can adapt to make throwing the exceptions easier.

For example, in `Quickstart`, if the required DBF setting which indicates where the database files are is missing, the following is used to throw an exception:

```
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:

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

This throws an `ErrorException` with a `DiagState` of `DIAG_GENERAL_ERROR` and the `QSInvalidCatalog` message key. `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 driver are looked up by their message key using an `IMessageSource` constructed by your Customer DSII Driver. 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 (installed by the SDK to `[INSTALL_DIRECTORY]\DataAccessComponents\ErrorMessages`). The driver 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 Unix/Linux platforms.

In order to provide messages of your own, you must register an error messages file with `DSIMessageSource` or construct your own `CustomerDSIIMessageSource` 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. See the SimbaEngine DSI API Reference Guide for full details on these classes and functions.

All of the sample drivers 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`. There are several message files that are used by the SDK components:

- **ODBCMessages.xml** – Contains the messages used by the common components of the SDK. This file must be present for all ODBC drivers.
- **SQLEngineMessages.xml** – Contains the messages used by the `SQLEngine`. This file only needs to be present for drivers using the `SQLEngine`.
- **CSCCommonMessages.xml** – Contains the messages common to both the `SimbaServer` and `SimbaClient` components. This file only needs to be present for `SimbaClient` installations, or for drivers using `SimbaServer`.
- **ServerMessages.xml** – Contains the messages specific to `SimbaServer`. This file only needs to be present for drivers using `SimbaServer`.
- **ClientMessages.xml** – Contains the messages specific to `SimbaClient`. This file only needs to be present for `SimbaClient` installations.

- `CLIDSIMessages.xml` – Contains the messages specific to the CLI DSI. This file only needs to be present for drivers using the CLI DSI.
- `JNIDSIMessages.xml` – Contains the messages specific to the JNI DSI. This file only needs to be present for drivers using the JNI DSI.

Custom SQL States

The Simba components strive to return SQL states, or equivalent, that are accurate for the specification they are implementing, ODBC for SimbaODBC, JDBC for SimbaJDBC, and ADO.NET for Simba.NET. However, in some cases your driver may need to return a different SQL state than that used by the SDK. In those cases, your DSI will return a custom SQL state by following one of the following cases.

ODBC

Exceptional cases are represented by exceptions, specifically 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

Exceptional cases are represented by exceptions, specifically 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

Exceptional cases are represented by exceptions, specifically 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.

Appendix C: Using Critical Section Locks

A critical section is a section of code that accesses a shared resource and this resource must not be accessed at the same time as another thread. For example, when writing to a log file, it would be ill advised to allow multiple threads to access this log file at the same time. If one thread writes to the log file while another thread writes to it, the logfile would not be in sync in each thread. The final result of both writes could be unpredictable – the lines that are written could be interleaved unexpectedly.

Critical sections of code where a shared resource is accessed (for example, logging to a log file), should be kept track of by using `CriticalSection` objects. A `CriticalSectionLock` object can then be used to lock this critical section so that no other thread can access the resource while another thread is executing. Please refer to the API Reference Guide for further details about `CriticalSection` and `CriticalSectionLock`.

For example, the function `CustomerDSIIDriver::GetDriverLog` should utilize a `CriticalSectionLock`.

You will need to include the following files:

```
#include "CriticalSection.h"  
#include "CriticalSectionLock.h"
```

1. Define a class member variable, `m_criticalSection`.

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

2. For functions that utilize shared resources, use a `CriticalSectionLock` to lock the critical section so that no other thread can use the resource before this function is finished with it. To do this, add the following to the start of the function:

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

The lock will be released once the function returns.

Appendix D: The Connection Process

Read this section for 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.

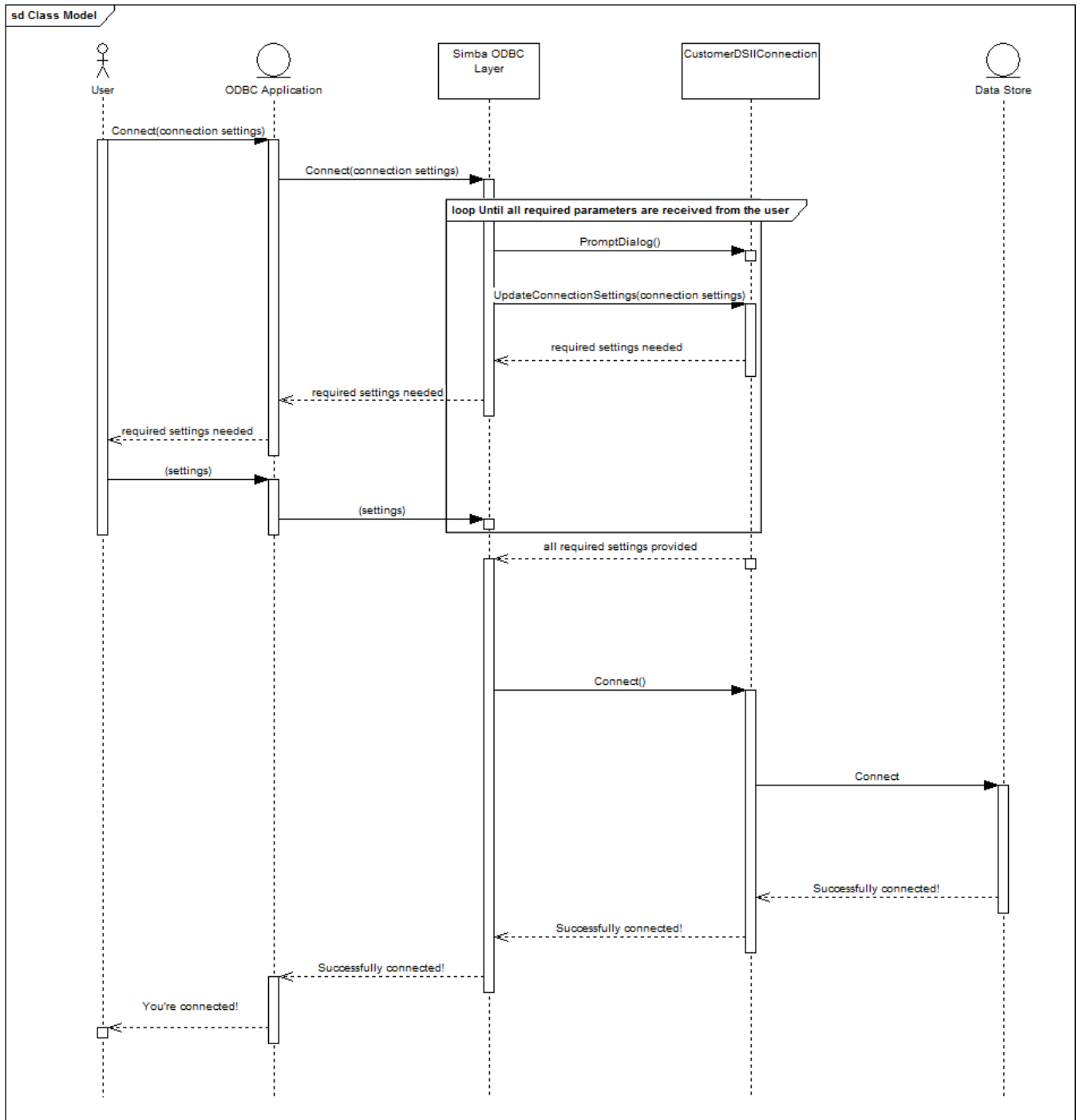


Figure 19: Sequence diagram illustrating the Connection Process.

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.

In your `CustomerDSIIConnection::UpdateConnectionSettings` function, the `in_connectionSettings` parameter will include 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.

There are utility functions `VerifyOptionalSetting` and `VerifyRequiredSetting` that can be used to simplify the process of checking if a setting exists and will put the appropriate value in the `out_connectionSettings` map if it does not. If you wish 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. There are two types of connection settings – those that are required, and those that are optional. This retrieve-request cycle repeats itself 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. There are utility functions `GetOptionalSetting` and `GetRequiredSetting` that can be used to simplify the process of extracting 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. 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. Refer to Appendix E: Creating and Using Dialogs for details on how to implement such a dialog box.

If you do not wish to implement a dialog box, you can simply leave the `PromptDialog` function empty.

Example Connection Process

Consider the call sequence if your DSII requires UID and PWD to establish a connection to your data store, and the application attempts a connection using `SQLDriverConnect` but only supplies the UID setting. First, `UpdateConnectionSettings` is called so that all the settings that are needed for a connection can be verified. Your `UpdateConnectionSettings` function would use `VerifyRequiredSetting` for both the UID and PWD 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 PWD key is missing, `out_connectionSettings` will have that setting added 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.

Appendix E: Creating and Using Dialogs

Windows

The Quickstart sample driver for Windows platforms includes a sample implementation of a user dialog. You have complete freedom to create dialogs of any type (or none at all) to allow your DSII to interact with users. A common use is to request connection settings. This appendix describes the `PromptDialog` function of the `CustomerDSIIConnection` class for displaying a dialog box that prompts the user for settings for this connection. The function `CustomerDSIIConnection::PromptDialog` has to be implemented. The function has the following declaration:

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

The parameters for the function are as follows:

- `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/MacOSX

Dialogs are also possible on Linux/Unix/MacOSX platforms, although our Quickstart sample driver 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 driver can make assumptions about running within a specific window system or API toolkit. Otherwise, the window you create will need to be parentless.

Appendix F: Posting Warnings

Warnings may be posted to an `IWarningListener`. The `DataStoreInterface` core classes `DSIEnvironment`, `DSIConnection` and `DSIStatement` each have an associated `IWarningListener`. This means your `CustomerDSIEnvironment`, `CustomerDSIConnection` and `CustomerDSIStatement` classes have access to an `IWarningListener`, which may be accessed through the parent `GetWarningListener` function.

The list of warnings that may be posted to an `IWarningListener` can be found in `$SIMBAENGINE_DIR\Include\Support\DiagState.h`

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. Refer to Appendix B: Errors and Exceptions for more information on how this mechanism operates.

To Subscribe to an `IWarningListener`:

Most of the classes with which warning listeners can be registered are controlled by the SDK and it isn't necessary for them to be explicitly registered by your DSII. The `ConnectionSetting` object is an exception to this and the warning listener must be registered after construction. See the example in Appendix D: The Connection Process, on page 120.

To Post Warnings to an `IWarningListener`:

We use the example of a `ConnectionSetting` subscribing to an `IWarningListener`.

```
// In CustomerDSIEnvironment, CustomerDSIConnection and
// CustomerDSIStatement, use the parent GetWarningListener()
// function to retrieve the IWarningListener
this->GetWarningListener()->PostWarning(
    Diagnostics::OPT_VAL_CHANGED,
    ComponentKey,
    L"WarningMessageKey");
```

Appendix G: Data Types

The data types associated with each of the SQL types are listed below for each of the C++, Java, and DotNet DSIs.

C++

SQL Type	Data Type
SQL_BIGINT (signed)	simba_int64
SQL_BIGINT (unsigned)	simba_uint64
SQL_BINARY	simba_byte*
SQL_BIT	simba_uint8
SQL_CHAR	simba_char*
SQL_DATE	TDWDate
SQL_DECIMAL	TDWExactNumericType
SQL_DOUBLE	simba_double64
SQL_FLOAT	simba_double64
SQL_INTEGER (signed)	simba_int32
SQL_INTEGER (unsigned)	simba_uint32
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_INTERVAL_MONTH	TDWSingleFieldInterval
SQL_INTERVAL_SECOND	TDWSecondInterval
SQL_INTERVAL_YEAR	TDWSingleFieldInterval
SQL_INTERVAL_YEAR_TO_MONTH	TDWYearMonthInterval
SQL_LONGVARBINARY	simba_byte*
SQL_LONGVARCHAR	simba_char*
SQL_NUMERIC	TDWExactNumericType
SQL_REAL	simba_double32

SQL_SMALLINT (signed)	simba_int16
SQL_SMALLINT (unsigned)	simba_uint16
SQL_TINYINT (signed)	simba_int8
SQL_TINYINT (unsigned)	simba_uint8
SQL_TIME	TDWTime
SQL_TIMESTAMP	TDWTimestamp
SQL_TYPE_DATE	TDWDate
SQL_TYPE_TIME	TDWTime
SQL_TYPE_TIMESTAMP	TDWTimestamp
SQL_VARBINARY	simba_byte*
SQL_VARCHAR	simba_char*
SQL_WCHAR	simba_byte*
SQL_WLONGVARCHAR	simba_byte*
SQL_WVARCHAR	simba_byte*

JDBC

Note that 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_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_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

C#

SQL Type	Data Type
SQL_BIGINT (signed)	System.Int64
SQL_BIGINT (unsigned)	System.UInt64
SQL_BINARY	array[System.Byte]
SQL_BIT	System.Boolean
SQL_CHAR	System.String
SQL_DECIMAL	System.Decimal or System.Data.SqlTypes.SqlDecimal
SQL_DOUBLE	System.Double
SQL_FLOAT	System.Double
SQL_INTEGER (signed)	System.Int32

SQL_INTEGER (unsigned)	System.UInt32
SQL_INTERVAL_DAY	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_DAY_TO_HOUR	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_DAY_TO_MINUTE	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_DAY_TO_SECOND	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_HOUR	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_HOUR_TO_MINUTE	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_HOUR_TO_SECOND	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_MINUTE	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_MINUTE_SECOND	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_MONTH	Simba.DotNetDSI.DataEngine.DSIMonthSpan
SQL_INTERVAL_SECOND	Simba.DotNetDSI.DataEngine.DSITimeSpan
SQL_INTERVAL_YEAR	Simba.DotNetDSI.DataEngine.DSIMonthSpan
SQL_INTERVAL_YEAR_TO_MONTH	Simba.DotNetDSI.DataEngine.DSIMonthSpan
SQL_LONGVARBINARY	array[System.Byte]
SQL_LONGVARCHAR	System.String
SQL_NUMERIC	System.Decimal or System.Data.SqlTypes.SqlDecimal
SQL_REAL	System.Single
SQL_SMALLINT (signed)	System.Int16
SQL_SMALLINT (unsigned)	System.UInt16
SQL_TINYINT (signed)	System.SByte
SQL_TINYINT (unsigned)	System.Byte
SQL_TYPE_DATE	System.DateTime
SQL_TYPE_TIME	System.DateTime
SQL_TYPE_TIMESTAMP	System.DateTime
SQL_VARBINARY	array[System.Byte]
SQL_VARCHAR	System.String
SQL_WCHAR	System.String
SQL_WLONGVARCHAR	System.String
SQL_WVARCHAR	System.String

Appendix H: Common Error Messages

This appendix lists some of the error messages you may encounter during the development and testing phases of your DSII.

A complete list of error codes and messages can be found by referring to the files in [INSTALL_DIRECTORY]\DataAccessComponents\ErrorMessages\.

Error Message	Meaning	Solution
The license file '<file>' could not be found. Please contact your administrator or Simba Technologies Inc. at support@simba.com.	The Simba.lic license file is not in the correct directory or you do not have a valid license.	Install the license file correctly by carefully following the instructions in Simba How-To #10100813 "Install Simba Evaluation Licence" available from http://www.simba.com/odbc-sdk-documents.htm
SQLDriverConnect returned: SQL_ERROR=-1	The Driver Manager cannot find or load the requested driver's DLL.	Make sure that your driver is installed correctly and that the DSN is correctly configured.
Specified driver could not be loaded.	The driver 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 driver 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. See section 1.6, "Contact Us" on page 10.
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 ErrorMessagePath 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.

Appendix 1: Build Platforms and compilers

This appendix lists which platforms and compilers are supported.

Platform	Version	Compiler
Windows	XP SP3	Visual Studio 2008
Windows	XP SP3	Visual Studio 2010
Linux (x86)	CentOS 4	GNU GCC 3.4.6
Linux (IA-64)	SLES 10	GNU GCC 4.1.2
Solaris (SPARC)	10	Sun C++ 5.8
Solaris (SPARC)	10	GNU GCC 3.4.3
Solaris (x86)	10	Sun C++ 5.8
Solaris (x86)	10	GNU GCC 3.4.3
AIX	5.3	IBM XL C/C++ Enterprise Edition V8.0
AIX	5.3	GNU GCC 4.2.4
HP-UX (IA-64)	11.23 (v2)	HP C/aC++ B3910B A.06.25
Mac OS X	10.6 (Snow Leopard)	GNU GCC 4.2.1

Appendix J: Driver Manager encodings on Linux/Unix/MacOSX

On non-Windows platforms, the driver configuration file needs to set the `DriverManagerEncoding` to indicate what type of Unicode is being passed to the driver 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
Mac OS X	32	UTF-32	
Mac OS X	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

Appendix K: Sample Unix Makefiles to build as a Shared Object

The example in this section assumes the following project directory structure:

```
<DSII>
  Source/
    Core/
    DataEngine/

  Bin/
```

Adjust the instructions to your own project directory structure as required. The instructions also assume the source file names that have been used throughout this documentation. Refer to the documentation for your compiler for details about each compiler option.

Build as a Shared Object example

Listed below is a sample Makefile to compile the CustomerDSII project. This example shows how to compile your driver as shared library with SQLEngine support.

```
#####
# Simba Technologies Inc.
# Copyright (C) 2011 Simba Technologies Incorporated
#
# File: Source/Makefile
#####

## -----
## PROJECT defines the project name. This is required for dependency inclusion.
## -----
PROJECT = CustomerDSII

## TARGET_SO is the full path and filename of the shared library.
TARGET_SO_PATH = ../Bin/$(PLATFORM)
TARGET_SO = $(TARGET_SO_PATH)/libCustomerDSII.so

#Path to the SimbaEngine libraries
SIMBA_LIB_PATH=$(SIMBAENGINE_DIR)/Lib/$(PLATFORM)

SIMBA_LIBS = \
$(SIMBA_LIB_PATH)/libDSI.a \
$(SIMBA_LIB_PATH)/libSimbaSupport.a \
$(SIMBA_LIB_PATH)/libSimbaODBC.a \
$(SIMBA_LIB_PATH)/libAEPprocessor.a \
$(SIMBA_LIB_PATH)/libCore.a \
$(SIMBA_LIB_PATH)/libDSIExt.a \
$(SIMBA_LIB_PATH)/libExecutor.a \
$(SIMBA_LIB_PATH)/libParser.a

## -----
## Common Sources used to build this project.
## -----
SRCS = Main.cpp \
Core/CustomerDSIIConnection.cpp \
Core/CustomerDSIIDriver.cpp \
Core/CustomerDSIIEnvironment.cpp \
Core/CustomerDSIIStatement.cpp \
DataEngine/CustomerDSIIDataEngine.cpp \
DataEngine/CustomerDSIIMetadataHelper.cpp \
DataEngine/CustomerDSIITable.cpp \
DataEngine/CustomerDSIITypeInfoMetadataSource.cpp \
```

```

## C sources
C_SRCS =
## Compiler flags
CFLAGS = \
-I. \
-I./Core \
-I./DataEngine \
-I$(SIMBAENGINE_DIR)/Include/DSI \
-I$(SIMBAENGINE_DIR)/Include/Support \
-I$(SIMBAENGINE_DIR)/Include/Support/Exceptions \
-I$(SIMBAENGINE_DIR)/Include/Support/TypedDataWrapper \
-I$(SIMBAENGINE_DIR)/Include/SQLEngine \
-I$(SIMBAENGINE_DIR)/Include/SQLEngine/AETree \
-I$(SIMBAENGINE_DIR)/Include/SQLEngine/DSIExt \
-I$(SIMBAENGINE_DIR)/ThirdParty/Expat \
-I$(SIMBAENGINE_DIR)/ThirdParty/iODBC/

# Definitions
RM = rm -f
RELEASE_C_OBJS = $(C_SRCS:.c=_$(PLATFORM)_release.c.o)
RELEASE_OBJS = $(SRCS:.cpp=_$(PLATFORM)_release.cpp.o) $(RELEASE_C_OBJS)

## Suffix Rules
.SUFFIXES: .cpp .c .o

## GLOBAL_CFLAGS is platform-specific.
%_$(PLATFORM)_release.c.o: %.c
$(CC) $(CFLAGS) $(GLOBAL_CFLAGS) -c $< -o $@

%_$(PLATFORM)_release.cpp.o: %.cpp
$(CXX) $(CFLAGS) $(GLOBAL_CFLAGS) -c $< -o $@

default: $(TARGET_SO)

clean:
$(RM) *.o
$(RM) DataEngine/*.o
$(RM) Core/*.o

## SO_LDFLAGS and LD are platform-specific.
$(TARGET_SO): $(RELEASE_OBJS) $(SIMBA_LIBS)
@echo "Building Shared Library $@: $(RELEASE_OBJS) $(SIMBA_LIBS)"
mkdir -p $(dir $(TARGET_SO))
$(LD) $(SO_LDFLAGS) $(RELEASE_OBJS) -o $@

```

AIX example using the IBM XL C/C++ compiler

Listed below is a sample Makefile to compile the CustomerDSII project using the IBM XL C/C++ compiler.

```

CXX      = xlc_r
CC       = xlc_r
LD       = xlc_r

## 64-bit
PLATFORM = AIX_powerpc64

GLOBAL_CFLAGS = -DSIMBA -D_REENTRANT -DNDEBUG -O2 -qmaxmem=-1 -q64 -qproto -qroconst -qrtti=all

SO_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -q64 -qmaxmem=-1 -bbigtoc -G
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -lc $(SIMBA_LIBS)

## 32-bit
PLATFORM = AIX_powerpc

GLOBAL_CFLAGS = -DSIMBA -D_REENTRANT -DNDEBUG -O2 -qmaxmem=-1 -q32 -qproto -qroconst -qrtti=all

```

```
SO_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -q32 -qmaxmem=-1 -bbigtoc -G
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -licui18n_simba32 -
licuuc_simba32
-lpthread -lm -lc $(SIMBA_LIBS)
```

AIX example using the GCC compiler

Listed below is a sample Makefile to compile the CustomerDSII project using the GCC compiler.

```
CXX      = g++
CC       = gcc
LD       = g++

## 64-bit
PLATFORM = AIX_powerpc64_gcc

GLOBAL_CFLAGS = -DSIMBA -D_REENTRANT -DNDEBUG -O2 -maix64 -fPIC -fexceptions

SO_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -maix64 -fPIC -Wl,-bbigtoc -Wl,-G
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -lc -Wl,-bM:SRE,-bnoentry,-bE:exports_AIX.map $(SIMBA_LIBS)

## 32-bit
PLATFORM = AIX_powerpc_gcc

GLOBAL_CFLAGS = -DSIMBA -D_REENTRANT -DNDEBUG -O2 -maix32 -fPIC -fexceptions

SO_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -maix32 -fPIC -Wl,-bbigtoc -Wl,-G
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -licui18n_simba32 -
licuuc_simba32
-lpthread -lm -lc -Wl,-bM:SRE,-bnoentry,-bE:exports_AIX.map $(SIMBA_LIBS)
```

AIX example exports file for compiling with the GCC compiler

Listed below is a sample exports_AIX.map file for compiling the CustomerDSII project using the GCC compiler.

```
#####
# Simba Technologies Inc.
# Copyright © 2011 Simba Technologies Incorporated
# AIX Exports file for compiling with GCC compiler
#####
SQLAllocHandle
SQLBindCol
SQLBindParameter
SQLBrowseConnect
SQLBrowseConnectW
SQLCancel
SQLCloseCursor
SQLColAttribute
SQLColAttributeW
SQLColumnPrivileges
SQLColumnPrivilegesW
SQLColumns
SQLColumnsW
SQLConnect
SQLConnectW
SQLCopyDesc
SQLDescribeCol
SQLDescribeColW
SQLDescribeParam
SQLDisconnect
SQLDriverConnect
SQLDriverConnectW
SQLEndTran
```

SQLExecDirect
SQLExecDirectW
SQLExecute
SQLExtendedFetch
SQLFetch
SQLFetchScroll
SQLForeignKeys
SQLForeignKeysW
SQLFreeHandle
SQLFreeStmt
SQLGetConnectAttr
SQLGetConnectAttrW
SQLGetCursorName
SQLGetCursorNameW
SQLGetData
SQLGetDescField
SQLGetDescFieldW
SQLGetDescRec
SQLGetDescRecW
SQLGetDiagField
SQLGetDiagFieldW
SQLGetDiagRec
SQLGetDiagRecW
SQLGetEnvAttr
SQLGetFunctions
SQLGetInfo
SQLGetInfoW
SQLGetStmtAttr
SQLGetStmtAttrW
SQLGetTypeInfo
SQLGetTypeInfoW
SQLMoreResults
SQLNativeSql
SQLNativeSqlW
SQLNumParams
SQLNumResultCols
SQLParamData
SQLPrepare
SQLPrepareW
SQLPrimaryKeys
SQLPrimaryKeysW
SQLProcedureColumns
SQLProcedureColumnsW
SQLProcedures
SQLProceduresW
SQLPutData
SQLRowCount
SQLSetConnectAttr
SQLSetConnectAttrW
SQLSetCursorName
SQLSetCursorNameW
SQLSetDescField
SQLSetDescFieldW
SQLSetDescRec
SQLSetEnvAttr
SQLSetPos
SQLSetStmtAttr
SQLSetStmtAttrW
SQLSpecialColumns
SQLSpecialColumnsW
SQLStatistics
SQLStatisticsW
SQLTablePrivileges
SQLTablePrivilegesW
SQLTables
SQLTablesW

Linux (x86) example

Listed below is a sample Makefile to compile the CustomerDSII project using the GCC compiler under the x86 architecture.

```
CXX          = g++
CC           = gcc
LD           = g++

## 64-bit
PLATFORM = Linux_x8664

GLOBAL_CFLAGS = -Wall -m64 -DSIMBA -D_REENTRANT -DCLUNIX -DNDEBUG -fPIC -O2

SO_LDFLAGS = -Wall -m64 -DSIMBA -D_REENTRANT -fPIC -O3 -shared
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -Wl,--whole-archive,$(SIMBA_LIBS) -Wl,--no-whole-archive -Wl,--version-
script=exports_Linux.map
-Wl,--soname=$(TARGET_SO)

## 32-bit
PLATFORM = Linux_x86

GLOBAL_CFLAGS = -Wall -m32 -DSIMBA -D_REENTRANT -DCLUNIX -DNDEBUG -fPIC -O2

SO_LDFLAGS = -Wall -m32 -DSIMBA -D_REENTRANT -fPIC -O3 -shared
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -licui18n_simba32 -
licuuc_simba32
-lpthread -lm -Wl,--whole-archive,$(SIMBA_LIBS) -Wl,--no-whole-archive -Wl,--version-
script=exports_Linux.map
-Wl,--soname=$(TARGET_SO)
```

Linux (Itanium) example

Listed below is a sample Makefile to compile the CustomerDSII project using the GCC compiler under the Itanium (ia64) 64-bit architecture.

```
CXX          = g++
CC           = gcc
LD           = g++

## 64-bit
PLATFORM = Linux_ia64

GLOBAL_CFLAGS = -Wall -DSIMBA -D_REENTRANT -DCLUNIX -DNDEBUG -fPIC -O2

SO_LDFLAGS = -Wall -DSIMBA -D_REENTRANT -fPIC -O3 -shared
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -Wl,--whole-archive,$(SIMBA_LIBS) -Wl,--no-whole-archive -Wl,--version-
script=exports_Linux.map
-Wl,--soname=$(TARGET_SO)
```

Linux example exports file for compiling with the GCC compiler

Listed below is a sample exports_Linux.map file for compiling the CustomerDSII project using the GCC compiler.

```
#####
# Simba Technologies Inc.
# Copyright © 2011 Simba Technologies Incorporated
# Linux Exports file.
```

```

#-----
{
global:
    create;
    destroy;
    SQLAllocHandle;
    SQLBindCol;
    SQLBindParameter;
    SQLBrowseConnect;
    SQLBrowseConnectW;
    SQLCancel;
    SQLCloseCursor;
    SQLColAttribute;
    SQLColAttributeW;
    SQLColumnPrivileges;
    SQLColumnPrivilegesW;
    SQLColumns;
    SQLColumnsW;
    SQLConnect;
    SQLConnectW;
    SQLCopyDesc;
    SQLDescribeCol;
    SQLDescribeColW;
    SQLDescribeParam;
    SQLDisconnect;
    SQLDriverConnect;
    SQLDriverConnectW;
    SQLEndTran;
    SQLExecDirect;
    SQLExecDirectW;
    SQLExecute;
    SQLExtendedFetch;
    SQLFetch;
    SQLFetchScroll;
    SQLForeignKeys;
    SQLForeignKeysW;
    SQLFreeHandle;
    SQLFreeStmt;
    SQLGetConnectAttr;
    SQLGetConnectAttrW;
    SQLGetCursorName;
    SQLGetCursorNameW;
    SQLGetData;
    SQLGetDescField;
    SQLGetDescFieldW;
    SQLGetDescRec;
    SQLGetDescRecW;
    SQLGetDiagField;
    SQLGetDiagFieldW;
    SQLGetDiagRec;
    SQLGetDiagRecW;
    SQLGetEnvAttr;
    SQLGetFunctions;
    SQLGetInfo;
    SQLGetInfoW;
    SQLGetStmtAttr;
    SQLGetStmtAttrW;
    SQLGetTypeInfo;
    SQLGetTypeInfoW;
    SQLMoreResults;
    SQLNativeSql;
    SQLNativeSqlW;
    SQLNumParams;
    SQLNumResultCols;
    SQLParamData;
    SQLPrepare;
    SQLPrepareW;
    SQLPrimaryKeys;
    SQLPrimaryKeysW;
    SQLProcedureColumns;
    SQLProcedureColumnsW;
    SQLProcedures;

```

```

    SQLProceduresW;
    SQLPutData;
    SQLRowCount;
    SQLSetConnectAttr;
    SQLSetConnectAttrW;
    SQLSetCursorName;
    SQLSetCursorNameW;
    SQLSetDescField;
    SQLSetDescFieldW;
    SQLSetDescRec;
    SQLSetEnvAttr;
    SQLSetPos;
    SQLSetStmtAttr;
    SQLSetStmtAttrW;
    SQLSpecialColumns;
    SQLSpecialColumnsW;
    SQLStatistics;
    SQLStatisticsW;
    SQLTablePrivileges;
    SQLTablePrivilegesW;
    SQLTables;
    SQLTablesW;
local: *;
};

```

HP-UX example on 64-bit Itanium architecture

Listed below is a sample Makefile to compile the CustomerDSII project on HP-UX under the Itanium (ia64) 64-bit architecture.

```

CXX      = /opt/aCC/bin/aCC
CC       = /opt/ansic/bin/cc
LD       = /opt/aCC/bin/aCC

PLATFORM = HP-UX_ia64

GLOBAL_CFLAGS = +Z -DPIC -DSIMBA -D_REENTRANT -D_THREAD_SAFE -D_LARGEFILE64_SOURCE
              -D_RWSTD_MULTI_THREAD -DNDEBUG +O2 +Ofltacc +Olibcalls +DD64 -AA -mt -ext +W495 +W740 +W749
              +W823 +W307 +W361 +W829 +W68 +W67 -n

SO_LDFLAGS = +Z -DPIC -DSIMBA -D_REENTRANT -D_THREAD_SAFE -D_LARGEFILE64_SOURCE
            -D_RWSTD_MULTI_THREAD +O2 +Ofltacc +Olibcalls -lrt +DD64 -AA -mt -lpthread -lc -lCsup -lstd_v2 -
            lunwind -ext +W495 +W740 +W749 +W823 +W307 +W361 +W829 +W68 +W67 -n -b
            -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
            licuuc_simba64
            -lpthread -lm -Wl,+forceload,$(SIMBA_LIBS) -Wl,+noforceload -Wl,+h=$(TARGET_SO)

```

Solaris example using Sun CC compiler on Sparc architecture

Listed below is a sample Makefile to compile the CustomerDSII project using the Sun CC compiler under the Sparc architecture.

```

CXX      = CC
CC       = cc
LD       = CC

## 64-bit
PLATFORM=Solaris_sparc64

GLOBAL_CFLAGS          = -DSIMBA -D_REENTRANT -DCLUNIX -DNDEBUG -xO5 -mt +w -xtarget=ultra3 -
xarch=v9a
-xcode=pic32 -D_POSIX_PTHREAD_SEMANTICS

```

```

SO_LDFLAGS = -DSIMBA -D_REENTRANT -xO5 -mt -zdefs +w -xtarget=ultra3 -xarch=v9a -xcode=pic32 -
lCrun -lCstd -lc -lrt -G -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64
-licui18n_simba64 -licuuc_simba64
-lpthread -lm -M exports_Solaris.map -zallextract $(SIMBA_LIBS) -zweakextract

## 32-bit
PLATFORM=Solaris_sparc

GLOBAL_CFLAGS          = -DSIMBA -D_REENTRANT -DCLUNIX -DNDEBUG -xO5 -mt +w -xtarget=ultra3 -
xarch=v8plusa
-xcode=pic32 -D_POSIX_PTHREAD_SEMANTICS

SO_LDFLAGS = -DSIMBA -D_REENTRANT -xO5 -mt -zdefs +w -xtarget=ultra3 -xarch=v8plusa -xcode=pic32
-lCrun -lCstd -lc -lrt -G -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32
-licui18n_simba32 -licuuc_simba32
-lpthread -lm -M exports_Solaris.map -zallextract $(SIMBA_LIBS) -zweakextract

```

Solaris example using Sun CC compiler on x86 architecture

Listed below is a sample Makefile to compile the CustomerDSII project using the Sun CC compiler under the x86 architecture.

```

CXX          = CC
CC           = cc
LD           = CC

## 64-bit
PLATFORM=Solaris_x8664

GLOBAL_CFLAGS          = -DSIMBA -D_REENTRANT -DCLUNIX -DNDEBUG -xO5 -mt +w -xtarget=opteron -
xarch=amd64
-KPIC -D_POSIX_PTHREAD_SEMANTICS

SO_LDFLAGS = -DSIMBA -D_REENTRANT -xO5 -mt -zdefs +w -xtarget=opteron -xarch=amd64 -KPIC -lCrun -
lCstd -lc -lrt
-G -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -M exports_Solaris.map -zallextract $(SIMBA_LIBS) -zweakextract

## 32-bit
PLATFORM=Solaris_x86

GLOBAL_CFLAGS          = -DSIMBA -D_REENTRANT -DCLUNIX -DNDEBUG -xO1 -mt +w -xtarget=opteron -KPIC
-D_POSIX_PTHREAD_SEMANTICS

SO_LDFLAGS = -DSIMBA -D_REENTRANT -xO1 -mt -zdefs +w -xtarget=opteron -KPIC -lCrun -lCstd -lc -
lrt
-G -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -licui18n_simba32 -
licuuc_simba32
-lpthread -lm -M exports_Solaris.map -zallextract $(SIMBA_LIBS) -zweakextract

```

Solaris example using GCC compiler on either Sparc or x86 architecture

Listed below is a sample Makefile to compile the CustomerDSII project using the GCC compiler under either the Sparc or x86 architecture.

```

CXX          = g++
CC           = gcc
LD           = g++

## Set the PLATFORM for the 64-bit target correctly depending on architecture.
## 64-bit Sparc
PLATFORM = Solaris_sparc64_gcc
## 64-bit x86
PLATFORM = Solarix_x8664_gcc

```

```

GLOBAL_CFLAGS          = -DSIMBA -D_REENTRANT -DCLUNIX -DNDEBUG -O2 -m64 -fPIC -
D_POSIX_PTHREAD_SEMANTICS

SO_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -m64 -fPIC -lc -lrt -shared
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -Wl,-M,exports_Solaris.map -Wl,-zallextract,$(SIMBA_LIBS) -Wl,-zweakextract

## Set the PLATFORM for the 32-bit target correctly depending on architecture.
## 32-bit Sparc
PLATFORM = Solaris_sparc_gcc
## 32-bit x86
PLATFORM = Solarix_x86_gcc

GLOBAL_CFLAGS          = -DSIMBA -D_REENTRANT -DCLUNIX -DNDEBUG -O2 -m32 -fPIC -
D_POSIX_PTHREAD_SEMANTICS

SO_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -m32 -fPIC -lc -lrt -shared
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -licui18n_simba32 -
licuuc_simba32
-lpthread -lm -Wl,-M,exports_Solaris.map -Wl,-zallextract,$(SIMBA_LIBS) -Wl,-zweakextract

```

Solaris example exports file for compiling with the GCC compiler

Listed below is a sample exports_Solaris.map file for compiling the CustomerDSII project using the GCC compiler.

```

=====
# Simba Technologies Inc.
# Copyright © 2011 Simba Technologies Incorporated
# Solaris Exports file.
=====
{
    local: *;
    global:
        SQLAllocHandle;
        SQLBindCol;
        SQLBindParameter;
        SQLBrowseConnect;
        SQLBrowseConnectW;
        SQLCancel;
        SQLCloseCursor;
        SQLColAttribute;
        SQLColAttributeW;
        SQLColumnPrivileges;
        SQLColumnPrivilegesW;
        SQLColumns;
        SQLColumnsW;
        SQLConnect;
        SQLConnectW;
        SQLCopyDesc;
        SQLDescribeCol;
        SQLDescribeColW;
        SQLDescribeParam;
        SQLDisconnect;
        SQLDriverConnect;
        SQLDriverConnectW;
        SQLEndTran;
        SQLExecDirect;
        SQLExecDirectW;
        SQLExecute;
        SQLExtendedFetch;
        SQLFetch;
        SQLFetchScroll;
        SQLForeignKeys;

```

```
SQLForeignKeysW;  
SQLFreeHandle;  
SQLFreeStmt;  
SQLGetConnectAttr;  
SQLGetConnectAttrW;  
SQLGetCursorName;  
SQLGetCursorNameW;  
SQLGetData;  
SQLGetDescField;  
SQLGetDescFieldW;  
SQLGetDescRec;  
SQLGetDescRecW;  
SQLGetDiagField;  
SQLGetDiagFieldW;  
SQLGetDiagRec;  
SQLGetDiagRecW;  
SQLGetEnvAttr;  
SQLGetFunctions;  
SQLGetInfo;  
SQLGetInfoW;  
SQLGetStmtAttr;  
SQLGetStmtAttrW;  
SQLGetTypeInfo;  
SQLGetTypeInfoW;  
SQLMoreResults;  
SQLNativeSql;  
SQLNativeSqlW;  
SQLNumParams;  
SQLNumResultCols;  
SQLParamData;  
SQLPrepare;  
SQLPrepareW;  
SQLPrimaryKeys;  
SQLPrimaryKeysW;  
SQLProcedureColumns;  
SQLProcedureColumnsW;  
SQLProcedures;  
SQLProceduresW;  
SQLPutData;  
SQLRowCount;  
SQLSetConnectAttr;  
SQLSetConnectAttrW;  
SQLSetCursorName;  
SQLSetCursorNameW;  
SQLSetDescField;  
SQLSetDescFieldW;  
SQLSetDescRec;  
SQLSetEnvAttr;  
SQLSetPos;  
SQLSetStmtAttr;  
SQLSetStmtAttrW;  
SQLSpecialColumns;  
SQLSpecialColumnsW;  
SQLStatistics;  
SQLStatisticsW;  
SQLTablePrivileges;  
SQLTablePrivilegesW;  
SQLTables;  
SQLTablesW;  
};
```

Appendix L: Building a driver for UnixODBC 2.2.12 and earlier

In order to get the driver to work properly with UnixODBC driver manager 2.2.12 and earlier, you need to use UnixODBC header files and link against the SimbaEngine static libraries that are build specifically for UnixODBC 2.2.12 and earlier. You need to make the following changes to the Makefile that is presented in the previous section.

SIMBA_LIBS

Change SIMBA_LIBS to the use the libraries with `_unixODBC` in their name

```
SIMBA_LIBS = \
$(SIMBA_LIB_PATH)/libDSI_unixODBC.a \
$(SIMBA_LIB_PATH)/libSimbaSupport_unixODBC.a \
$(SIMBA_LIB_PATH)/libSimbaODBC_unixODBC.a \
$(SIMBA_LIB_PATH)/libAEProcessor_unixODBC.a \
$(SIMBA_LIB_PATH)/libCore_unixODBC.a \
$(SIMBA_LIB_PATH)/libDSIExt_unixODBC.a \
$(SIMBA_LIB_PATH)/libExecutor_unixODBC.a \
$(SIMBA_LIB_PATH)/libParser_unixODBC.a
```

CFLAGS

Change CFLAGS to use unixODBC header files. Also, you need to use the correct preprocessor definition whether your driver is 64-bit or 32-bit.

```
CFLAGS = \
-I. \
-I./Core \
-I./DataEngine \
-I$(SIMBAENGINE_DIR)/Include/DSI \
-I$(SIMBAENGINE_DIR)/Include/DSI/Client \
-I$(SIMBAENGINE_DIR)/Include/Support \
-I$(SIMBAENGINE_DIR)/Include/Support/Exceptions \
-I$(SIMBAENGINE_DIR)/Include/Support/TypedDataWrapper \
-I$(SIMBAENGINE_DIR)/Include/SQLEngine \
-I$(SIMBAENGINE_DIR)/Include/SQLEngine/AETree \
-I$(SIMBAENGINE_DIR)/Include/SQLEngine/DSIExt \
-I$(SIMBAENGINE_DIR)/Include/Server \
-I$(SIMBAENGINE_DIR)/ThirdParty/Expat \
-I$(SIMBAENGINE_DIR)/ThirdParty/unixODBC/

## 64-bit
CFLAGS := $(CFLAGS) -DUNIXODBC -DSIZEOF_LONG=8 -DSQL_WCHART_CONVERT

## 32-bit
CFLAGS := $(CFLAGS) -DUNIXODBC -DSIZEOF_LONG=4 -DSQL_WCHART_CONVERT
```

Appendix M: Sample Unix Makefiles to build as a SimbaServer

The example in this section assumes the following project directory structure:

```
<DSII>
  Source/
    Core/
    DataEngine/

  Bin/
```

Adjust the instructions to your own project directory structure as required. The instructions also assume the source file names that have been used throughout this documentation. Refer to the documentation for your compiler for details about each compiler option.

Build as a SimbaServer example

Listed below is a sample Makefile to compile the CustomerDSII project. This example shows how to compile your driver as a SimbaServer with SQLEngine support.

NOTE: The GLOBAL_CFLAGS, CC, CXX, LD, and PLATFORM are the same for building the driver as a server and shared library. Therefore, please look at the previous section to find the correct flags for GLOBAL_CFLAGS for your platform.

```
#####
# Simba Technologies Inc.
# Copyright (C) 2011 Simba Technologies Incorporated
#
# File: Source/Makefile
#####

## -----
## PROJECT defines the project name. This is required for dependency inclusion.
## -----
PROJECT = CustomerDSII

## TARGET_BIN is the full path and filename of the server.
TARGET_BIN_PATH = ../Bin/$(PLATFORM)
TARGET_BIN = $(TARGET_BIN_PATH)/CustomerDSIIServer

#Path to the SimbaEngine libraries
SIMBA_LIB_PATH=$(SIMBAENGINE_DIR)/Lib/$(PLATFORM)

SIMBA_LIBS = \
$(SIMBA_LIB_PATH)/libDSI.a \
$(SIMBA_LIB_PATH)/libSimbaSupport.a \
$(SIMBA_LIB_PATH)/libAEProcessor.a \
$(SIMBA_LIB_PATH)/libCore.a \
$(SIMBA_LIB_PATH)/libDSIExt.a \
$(SIMBA_LIB_PATH)/libExecutor.a \
$(SIMBA_LIB_PATH)/libParser.a \
$(SIMBA_LIB_PATH)/libSimbaCommunications.a \
$(SIMBA_LIB_PATH)/libSimbaMessages.a \
$(SIMBA_LIB_PATH)/libSimbaServer.a

## -----
## Common Sources used to build this project.
## -----
```

```

SRCS = Main.cpp \
Core/CustomerDSIIConnection.cpp \
Core/CustomerDSIIDriver.cpp \
Core/CustomerDSIEnvironment.cpp \
Core/CustomerDSIIStatement.cpp \
DataEngine/CustomerDSIIDataEngine.cpp \
DataEngine/CustomerDSIIMetadataHelper.cpp \
DataEngine/CustomerDSIITable.cpp \
DataEngine/CustomerDSIITypeInfoMetadataSource.cpp \

## C sources
C_SRCS =

## Compiler flags
CFLAGS = \
-I. \
-I./Core \
-I./DataEngine \
-I./DataEngine/Metadata \
-I$(SIMBAENGINE_DIR)/Include/DSI \
-I$(SIMBAENGINE_DIR)/Include/DSI/Client \
-I$(SIMBAENGINE_DIR)/Include/Support \
-I$(SIMBAENGINE_DIR)/Include/Support/Exceptions \
-I$(SIMBAENGINE_DIR)/Include/Support/TypedDataWrapper \
-I$(SIMBAENGINE_DIR)/Include/Server \
-I$(SIMBAENGINE_DIR)/ThirdParty/Expat

# Definitions
RM = rm -f
RELEASE_C_OBJS = $(C_SRCS:.c=_$(PLATFORM)_release.c.o)
RELEASE_OBJS = $(SRCS:.cpp=_$(PLATFORM)_release.cpp.o) $(RELEASE_C_OBJS)

## Suffix Rules
.SUFFIXES: .cpp .c .o

## GLOBAL_CFLAGS is platform-specific and it's the same as building the driver as shared library.
%_$(PLATFORM)_release.c.o: %.c
    $(CC) $(CFLAGS) $(GLOBAL_CFLAGS) -c $< -o $@

%_$(PLATFORM)_release.cpp.o: %.cpp
    $(CXX) $(CFLAGS) $(GLOBAL_CFLAGS) -c $< -o $@

default: $(TARGET_BIN)

clean:
    $(RM) *.o
    $(RM) DataEngine/*.o
    $(RM) Core/*.o

## BIN_LDFLAGS and LD are platform-specific.
## LD is the same as building the driver as shared library.
$(TARGET_BIN): $(RELEASE_OBJS) $(SIMBA_LIBS)
    @echo "Building Binary $@: $(RELEASE_OBJS) $(SIMBA_LIBS)"
    mkdir -p $(dir $(TARGET_BIN))
    $(LD) $(BIN_LDFLAGS) $(RELEASE_OBJS) -o $@

```

AIX example using the IBM XL C/C++ compiler

Listed below is a sample Makefile to compile the CustomerDSII project using the IBM XL C/C++ compiler.

```

## 64-bit
BIN_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -q64 -qmaxmem=-1 -bbigtoc -brtl
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licuil8n_simba64 -
licuuc_simba64
-lpthread -lm -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto -lnsl -lc
,$(SIMBA_LIBS)

## 32-bit

```

```

BIN_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -q32 -qmaxmem=-1 -bbigtoc -brtl
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -licui18n_simba32 -
licuuc_simba32
-lpthread -lm -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto -lnsl -lc
,$(SIMBA_LIBS)

```

AIX example using the GCC compiler

Listed below is a sample Makefile to compile the CustomerDSII project using the GCC compiler.

```

## 64-bit
BIN_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -maix64 -fPIC -Wl,-bbigtoc -Wl,-brtl
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto -lnsl -lc
,$(SIMBA_LIBS)

## 32-bit
BIN_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -maix32 -fPIC -Wl,-bbigtoc -Wl,-brtl
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -licui18n_simba32 -
licuuc_simba32
-lpthread -lm -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto -lnsl -lc
,$(SIMBA_LIBS)

```

Linux (x86) example

Listed below is a sample Makefile to compile the CustomerDSII project using the GCC compiler under the x86 architecture.

```

## 64-bit
BIN_LDFLAGS = -Wall -m64 -DSIMBA -D_REENTRANT -fPIC -O3
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto
-Wl,--whole-archive,$(SIMBA_LIBS) -Wl,--no-whole-archive

## 32-bit
BIN_LDFLAGS = -Wall -m32 -DSIMBA -D_REENTRANT -fPIC -O3
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto
-Wl,--whole-archive,$(SIMBA_LIBS) -Wl,--no-whole-archive

```

Linux (Itanium) example

Listed below is a sample Makefile to compile the CustomerDSII project using the GCC compiler under the Itanium (ia64) 64-bit architecture.

```

## 64-bit
BIN_LDFLAGS = -Wall -DSIMBA -D_REENTRANT -fPIC -O3
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto
-Wl,--whole-archive,$(SIMBA_LIBS) -Wl,--no-whole-archive

```

HP-UX example on 64-bit Itanium architecture

Listed below is a sample Makefile to compile the CustomerDSII project on HP-UX under the Itanium (ia64) 64-bit architecture.

```

BIN_LDFLAGS = +Z -DPIC -DSIMBA -D_REENTRANT -D_THREAD_SAFE -D_LARGEFILE64_SOURCE
-D_RWSTD_MULTI_THREAD +O2 +Ofltacc +Olibcalls -lrt +DD64 -AA -mt -lpthread -lc -lCsup -lstd_v2 -
lunwind -ext +W495 +W740 +W749 +W823 +W307 +W361 +W829 +W68 +W67 -n
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto -
Wl,+forceload,$(SIMBA_LIBS)
-Wl,+noforceload

```

Solaris example using Sun CC compiler on Sparc architecture

Listed below is a sample Makefile to compile the CustomerDSII project using the Sun CC compiler under the Sparc architecture.

```

## 64-bit
BIN_LDFLAGS = -DSIMBA -D_REENTRANT -xO5 -mt -zdefs +w -xtarget=ultra3 -xarch=v9a -xcode=pic32 -
lCrun -lCstd -lc -lrt -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -
licui18n_simba64 -licuuc_simba64
-lpthread -lm -lsocket -lnsl -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto
-zallextract $(SIMBA_LIBS) -zweakextract

## 32-bit
SO_LDFLAGS = -DSIMBA -D_REENTRANT -xO5 -mt -zdefs +w -xtarget=ultra3 -xarch=v8plusa -xcode=pic32
-lCrun -lCstd -lc -lrt -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -
licui18n_simba32 -licuuc_simba32
-lpthread -lm -lsocket -lnsl -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto
-zallextract $(SIMBA_LIBS) -zweakextract

```

Solaris example using Sun CC compiler on x86 architecture

Listed below is a sample Makefile to compile the CustomerDSII project using the Sun CC compiler under the x86 architecture.

```

## 64-bit
SO_LDFLAGS = -DSIMBA -D_REENTRANT -xO5 -mt -zdefs +w -xtarget=opteron -xarch=amd64 -KPIC -lCrun -
lCstd -lc -lrt
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64
-lpthread -lm -lsocket -lnsl -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto
-zallextract $(SIMBA_LIBS) -zweakextract

## 32-bit
SO_LDFLAGS = -DSIMBA -D_REENTRANT -xO1 -mt -zdefs +w -xtarget=opteron -KPIC -lCrun -lCstd -lc -
lrt
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -licui18n_simba32 -
licuuc_simba32
-lpthread -lm -lsocket -lnsl -L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto
-zallextract $(SIMBA_LIBS) -zweakextract

```

Solaris example using GCC compiler on either Sparc or x86 architecture

Listed below is a sample Makefile to compile the CustomerDSII project using the GCC compiler under either the Sparc or x86 architecture.

```

## 64-bit for both x86 and Sparc architecture
SO_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -m64 -fPIC -lc -lrt -
L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba64 -licui18n_simba64 -
licuuc_simba64 -lpthread -lm -lsocket -lnsl
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto
-Wl,-zallextract,$(SIMBA_LIBS) -Wl,-zweakextract

```

```
## 32-bit for both x86 and Sparc architecture
SO_LDFLAGS = -DSIMBA -D_REENTRANT -O2 -m32 -fPIC -lc -lrt -
L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -licudata_simba32 -licui18n_simba32 -
licuuc_simba32 -lpthread -lm -lsocket -lnsl
-L$(SIMBAENGINE_DIR)/ThirdParty/icu/$(PLATFORM)/lib -lssl -lcrypto
-Wl,-zallextract,$(SIMBA_LIBS) -Wl,-zweakextract
```

Appendix N: Localization

There are several ways in which the end-user can configure the locale:

3. A locale for the driver (driver-wide) can be configured. If the locale is not configured, the default locale for the SDK (en-US) is used.
4. A locale can be configured for each connection. This is referred to as a per-connection locale. If the locale is not configured for the connection, then the driver-wide locale is used.

Localizing your driver can be simplified by using the same format for error messages as the SDK.

Locales

Locale values are composed of 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 separate from the language code by a hyphen (-).

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

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)

ODBC

The ODBC Error Messages are divided into several smaller files. The table below describes each file, and explains which error message files must be included when you distribute your driver.

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 driver. If you distribute SimbaClient for ODBC, you will also need to include this file.
SQLEngineMessages.xml	Contains the error messages for the SimbaEngine components.	Only if your driver uses SimbaEngine.
ClientMessages.xml	Contains the error messages for SimbaClient for ODBC.	Only if you are distributing SimbaClient for ODBC.
CSCCommonMessages.xml	Contains the error messages for the Client/Server protocol components.	Only if you have built your driver 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 driver as a server.
CLIDSIMessages.xml	Contains the error messages for the CLIDSI component.	Only if your driver uses the CLIDSI component.
JNIDSIMessages.xml	Contains the error messages for the JNIDSI component.	Only if your driver uses the JNIDSI component.

Localizing Your ODBC Driver

The SDK currently only supports the English – United States (en-US) locale. However, it exposes a mechanism, which you can use to easily add support for additional locales.

Two common conventions are supported for organizing localized error message files.

- Each locale’s message files are stored in a subdirectory, with the name of the locale.

```
Example :
ErrorMessages
  en-US
    ODBCMessages.xml
  fr-CA
    ODBCMessages.xml
```

```
ja
  ODBCMessages.xml
```

- All message files, for every locale is stored in a single folder. The name of the locale is a suffix for the file.

```
ErrorMessages
  ODBCMessages_en-US.xml
  ODBCMessages_fr-CA.xml
  ODBCMessages_ja.xml
```

JDBC

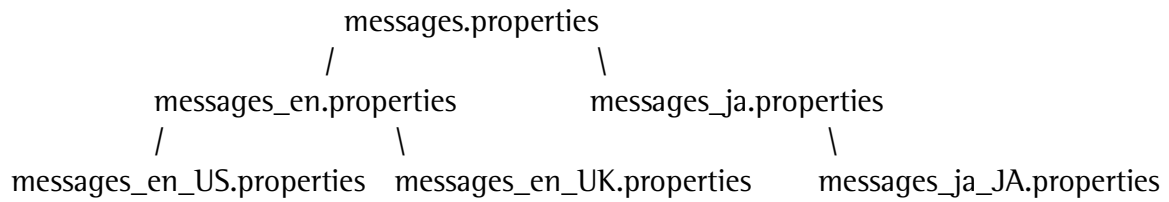
The JDBC Error Messages are divided into several smaller files. The table below describes each file, and explains which error message files must be included when you distribute your driver.

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 driver. 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 driver. 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.

Localizing Your JDBC Driver

The SDK currently only supports the English – United States (en-US) locale. However, it exposes a mechanism, which you can use to easily add support for additional locales.

The common convention for localization with ResourceBundles is used to organize the error message files. Each message file should be defined in a hierarchy to ensure that even if a locale is not supported, a parent message file will be used. For example, the message file hierarchical structure could be:



In this example, "messages" is the base file name. Note that each message file would need to be registered separately with the DSIMessageSource.

Appendix O: Supported ODBC scalar functions

This version of the SimbaEngine SDK supports the following ODBC defined scalar functions:

Explicit Covert function:

- CONVERT

String Functions:

- CONCAT
- INSERT
- LCASE
- LEFT
- LENGTH
- LOCATE
- LTRIM
- REPLACE
- RIGHT
- RTRIM
- SPACE
- SUBSTRING
- UCASE

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

Time, Date, and Interval Functions:

- CURDATE
- CURTIME
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- DAYNAME
- DAYOFMONTH
- DAYOFWEEK
- DAYOFYEAR
- HOUR
- MINUTE
- MONTH
- MONTHNAME
- NOW
- QUARTER
- SECOND
- TIMESTAMPDIFF
- WEEK
- YEAR

System Functions:

- DATABASE
- IFNULL
- USER