insightsoftware

Simba SDK

Build a JDBC Driver for a SQL-Capable Data Store in 5 Days

Version 10.3

August 2024

Copyright

Contact Us

insightsoftware

www.insightsoftware.com

Table of Contents

About this Guide

Purpose

This guide explains how to use the Simba SDK to create a Type 4 (100% pure JDBC) connector for a data store that is SQL-aware. It explains how to customize the JavaUltraLight sample connector, which is included with the Simba SDK.

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

Note:

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

Advantages of Using the Simba SDK

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

The Simba SDK, developed by experts in the field, encapsulates all the required functionality and exposes an easy-to-use SDK that allows you to create a robust and efficient database connector for your data store.

Build a Custom JDBC Connector in Five Days

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

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

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

Audience

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

Document Conventions

Italics are used when referring to book and document titles.

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

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

NOTE:

Indicates a short note appended to a paragraph.

Important:
IMPORTANT:

Indicates an important comment related to the preceding paragraph.

Knowledge Prerequisites

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

- Familiarity with the Java programming language.
- Ability to use the data store to which the connector you are developing will connect.
- An understanding of the role of JDBC technologies in connecting to a data store.
- Exposure to SQL.

Variables Used in this Document

The following variables are used in this document:

Variable	Description
[INSTALL_DIR]	Installation directory for the SimbaEngine X SDK. **Default value on Windows platforms**: `C:\insightsoftware\SimbaEngineSDK\10.3` **Default value on Linux, Unix, and macOS platforms**: `[UNTAR_DIR]/SimbaEngineSDK/10.3`

Variable	Description
[UNTAR_DIR]	Directory where the SimbaEngine X SDK distributable was untarred.
[JDBC_VERSION]	The version of JDBC that your driver supports. You can use the SimbaEngine X SDK to build a driver for version 4.2 and 4.3, or a hybrid version. Possible values of *[JDBC_VERSION]* are `42` and `43`, and `Hybrid`.

Introduction

This guide will show you how to create a custom Type 4 JDBC driver using SimbaEngine. It will walk you through the steps to modify and customize the included JavaUltraLight sample driver. Full documentation for SimbaEngine is available on the Simba website at http://www.simba.com/odbc-sdk-documents.htm.

Conventions Used in this Manual

This document is primarily focused on a Windows-based development environment. However, at selected key points, significant differences relating to Linux environments are highlighted in blue-tinted text blocks like this one.

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

The libraries containing the data access components you need are in the `[INSTALL_DIRECTORY]\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.

Function names, SQL keywords, and program names appear in narrow bold type when described in text. For example:

Implement the constructor `CustomerDSIIConnection::CustomerDSIIConnection`

The full path of the installation directory for SimbaEngine may vary depending on your system. In this document, we will represent it as `[INSTALL_DIRECTORY]`, which defaults as follows (note that “`10.3`” is the release and this part of the path will change with each release of SimbaEngine):

- Windows platforms:

  C:\Simba Technologies\SimbaEngineSDK\

- Linux/Unix/MacOSX platforms:

  <theUntarDirectory>/SimbaEngineSDK/

For More Information

The following documentation is available for SimbaEngine:

- **SimbaEngine Developer Guide**: Detailed information on how to work with SimbaEngine 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 SimbaEngine, 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 SimbaEngine, 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 SimbaEngine, using Java as your development environment.

- **Build a JDBC Driver for SQL-Based Data Sources in 5 Days**: (this document) Condensed information to walk you through the process of creating a custom Type 4 JDBC driver with SimbaEngine, 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 SimbaEngine, 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 server and client components of SimbaEngine.

- **SimbaEngine Release Notes**: Information about incremental changes introduced in a particular release of SimbaEngine.

For complete information on the JDBC 4.2 API specification, see the `java.sql` and `javax.sql` packages in the Java Platform, Standard Edition 8 API Specification, available from the Oracle website at: http://docs.oracle.com/javase//docs/api/

For complete information on the JDBC 4.2 API specification, see the java.sql and javax.sql packages in the Java Platform, Standard Edition API Specification, available from the Oracle website at: http://docs.oracle.com/javase//docs/api/

Getting Started

This document describes the steps required to build a prototype JDBC driver on Windows using SimbaEngine to access your data store.

Figure 1: High level view of SimbaEngine

SimbaEngine provides implementations of the JDBC 4.4. specifications (all of required features and most of the optional features) which provides standard interfaces to which any JDBC enabled application can connect.

The libraries of SimbaEngine hide all of the complexity of error checking, session management, data conversions and other low-level implementation details. They expose a simple API (called the Data Store Interface API or DSI API), which defines the primitive operations needed to access a data store. As an SDK developer, you will create an implementation of a DSI (also known as a DSI Implementation or DSII) that will access your particular data source.

The sample projects provided with SimbaEngine are functioning DSI Implementations that you can copy and modify for your own purposes. The JavaUltraLight example that is the primary subject of this document connects to an in-memory table. Following the steps in this guide, you will change the JavaUltraLight example driver so that it accesses your own data store instead of the example data store included with SimbaEngine.

What You are Working Toward

The SimbaEngine libraries allow you to create data access solutions that connect to SQL or non-SQL data sources that are either local or remote to the end users' computers. All of the possible SimbaEngine implementation architectures are discussed in more detail in the SimbaEngine Developer Guide, but they are explained briefly here for context.

If you plan for your users to connect to your data source locally, you will compile and link your driver as a JAR, then install and register that on each computer containing a data source. If you plan to connect your users to a network-based data source, you will link your driver with the Simba Client/Server libraries (with no changes to your DSII code) to create a stand-alone SimbaServer executable that will run on your database server. You will then distribute the generic SimbaJDBCClient to individual users. The prototype that you will build by following this document accesses a local data store. Please see the SimbaClientServer User Guide for more details on remote database configurations.

Since SimbaEngine provides the flexibility to build a driver for JDBC 4.3 or 4.2, Simba recommends that your resulting binary (i.e. .jar or .exe) be named accordingly to indicate the target version it is intended for. For example, the JavaUltraLight project outputs `JavaUltraLight_43.jarJavaUltraLight_42.jar` for the respective targets.

How You Will Get There

In the SimbaEngine JavaUltraLight driver code we have highlighted the areas you need to change by adding comments prefixed with "`TODO`" so you can find them, along with a short explanatory message. There are eight areas in the code highlighted with `TODOs` and this document will walk you through each of them.

Of the areas of the code that you need to modify, most are for productization rather than actually connecting your data store to SimbaEngine. These are things like naming the driver, setting the properties that configure the driver, and naming the error file and log files. In other words, these are not complicated tasks.

The remaining areas of the code that you will modify are primarily concerned with connecting to your data store, preparing and executing queries, and getting the data and metadata from your data store into SimbaEngine. Since the JavaUltraLight driver already has the classes and code to do this against the example data store, all you have to do is modify what already exists and your driver will begin to work against your own data store.

The simplicity of the DSI API also helps you because there is order and symmetry to the way it works. The UML diagram below shows the design pattern to look for. Almost all DSI implementations wind up with a similar pattern. Look for the pattern of class relationships headed by `IDataEngine, IQueryExecutor` and `IResultSet` and anchored by your `DataEngine, QueryExecutor, MetadataSource` and `Table classes`, e.g. `ULDataEngine`, `ULQueryExecutor`, `TablesMetadataSource` and `ULPersonTable` in the JavaUltraLight Driver. If your resultant DSI implementation design looks like this, you are on the right track.

Figure 1: Design pattern for a DSI implementation.

Implementing data retrieval is straightforward. Your Reader class interacts directly with your data store to retrieve the data and deliver it to the Table class on demand. The Reader class should take care of caching, buffering, paging, and all the other techniques that speed data access. Implementing metadata access is a bit more complicated, but it is not as bad as it looks. There are several Metadata Sources that you can implement, but as a starting point, to make your driver work properly you only need to implement these five Metadata Sources:

1. Catalog only
2. Catalog/Schema only
3. Columns
4. Tables
5. Type Information

SimbaEngine also includes the JavaQuickStart example. This example is primarily used as the starting point for writing an ODBC driver using Java as the language platform for the DSII. It currently cannot be used as the starting point for a Type 4 JDBC driver.

What's Included in SimbaEngine

The default `[INSTALL_DIRECTORY]` is:

- Windows platforms: `C:\Simba Technologies\SimbaEngineSDK\10.3`
- Linux/Unix/MacOSX platforms: `<theUntarDirectory>/SimbaEngineSDK/10.3`

  For a JDBC driver, important files are located in the directories below:

- `[INSTALL_DIRECTORY]\DataAccessComponents\Lib`:

  The SimbaJDBC.jar and SimbaJDBC-javadoc.jar files are found here.

- `[INSTALL_DIRECTORY]\Examples\Source`:

  Sub-folders containing all the source code and project files to create complete drivers that will work against provided sample data. The purpose of these examples is to provide complete solutions both so you can see how your custom solution will look when you are done, and actually provide a starting point for your driver.

  `\JavaUltraLight` sub-folder: Source code and project files of the sample JavaUltraLight driver for immediate use and reference.

  `\Lib` sub-folder: Compiled JavaUltraLight driver JAR file for immediate use.

  The name of the driver will vary depending on which version of JDBC (4.3 or 4.2) the driver is being developed for.

- `[INSTALL_DIRECTORY]\Documentation\SSLCertificates`: Simba self-signed example SSL certificates.

Libraries Included With SimbaEngine

SimbaEngine includes libraries for developing JDBC drivers, ADO.NET providers and ODBC drivers using C++, Java, or C#. This document is concerned with the library used to develop JDBC drivers.

The `\Lib` sub-folder contains the Release versions of the Simba JDBC .jar files which contain the JDBC and DSI components for a JDBC driver. Depending on which version of JDBC you're developing your driver for (4.3 or 4.2), you will use the respective .jar file from this folder: `SimbaJDBC_4.jarSimbaJDBC_.jar`.

Build a JDBC Driver in Five Days

SimbaEngine ships with a sample driver that you can use for the basis of your own drivers. Over the course of the 5-day plan, you will modify the JavaUltraLight sample driver so that it accesses your own data store instead of the example in-memory data store. Here is a quick overview of what you will be doing each day:

Day	Activities
One	▪ Install SimbaEngine and build the sample driver included with SimbaEngine. ▪ Test the sample driver using a JDBC-enabled application. ▪ Set up a new project directory where you will begin to modify the sample driver as the starting point for your new driver.
Two	▪ Set the driver name. ▪ Set the driver and connection properties. ▪ Set the driver-wide and connection-wide logging level. ▪ Modify the connection setting validation method to match the connection string needed for your data store. ▪ Modify the connection authentication method to match the settings needed for your data store.
Three	▪ Modify the method used to create and return your Metadata Sources, which will support JDBC metadata methods. ▪ Modify the class used to support the getTypeInfo JDBC metadata method with the data types supported by your data store. ▪ Modify the other classes used to support the JDBC database metadata methods needed by the majority of JDBC-enabled applications.
Four	▪ Modify the method used to prepare a query for your data store. ▪ Modify the methods used to execute a query for your data store. ▪ Modify the methods used to navigate through and retrieve data from your data store.

Day	Activities
Five	▪ Modify the display names and identifiers appropriate for your organization and driver. ▪ Modify the parameters of the exceptions thrown by your driver to match as well. ▪ Rename the packages, files and classes to use a name and abbreviation appropriate for your driver.

Day One

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

Initial Set Up

Start by installing SimbaEngine and running the example driver:

1. Install SimbaEngine using the setup executable – run this program and follow the instructions of the installer.

   If you have previously installed a version of SimbaEngine, uninstall it before installing the new one. Also, note that SimbaEngine environment variables are defined only for the user that ran the installation. If you install SimbaEngine as a regular user and then run Visual Studio as an administrator, SimbaEngine will not work properly.

2. You are now ready to build, test and verify that the example driver is working. You can use any JDBC application for testing, such as Crystal Reports, DbVisualizer or SQuirreL.

On Linux platforms, SimbaEngine is provided as a single file consisting of the SimbaEngineSDK*.tar.gz file, a tar format archive that has been compressed using the gzip tool (where the "*" represents a string of alphanumeric characters that represent the build number and platform of the kit). On these platforms, the installation steps are as follows:

1. Open a command prompt and change to a directory where you would like to install SimbaEngine.

2. To uncompress `SimbaEngineSDK*.tar.gz`, enter:
   ```
   gzip -d SimbaEngineSDK*.tar.gz
   ```
   This will extract the file Simba.tar.

3. To install SimbaEngine enter:
   ```
   tar -xvf SimbaEngineSDK*.tar.
   ```

Build and Test the Example Driver

1. The source for the example JDBC driver is located in the Examples folder. These instructions assume you will be working with the Eclipse integrated development environment (IDE) with the version of JDK being 1.9 or 1.8. Ensure that you have the environment variable JAVA_HOME pointing to the root of your JDK.

2. Use Eclipse to import the desired JavaUltraLight project as an existing project into your workspace. The DSII project files are located under `[INSTALL_ DIRECTORY]\Examples\Source\JavaUltraLight\Source\ JavaUltraLightDSII\ [JDBC_VERSION]`.

   - From the Main Menu, select **Project->Properties->Java Build Path->Libraries**.
   - Select the **SIMBAENGINE_DIR** entry, **Edit**, **Variable**, **New** and create a new classpath variable named SIMBAENGINE_DIR with the Path pointing to the DataAccessComponents folder of your installed SimbaEngine.
   - Select **OK** and perform a full rebuild of the workspace when prompted. When you see that there are no errors, you are ready to build the driver.

3. Building against your version of JDBC:

   - From the **Package Explorer**, expand the **JavaUltraLight** project.
   - Right-click on the `JavaUltraLightBuilder[43 or 42].xml` file.
   - Select **Run As**.
   - Click on 'Ant Build…'. On the Targets tab select **JavaUltraLightBuildDebug[43 or 42]**, depending on which version of JDBC the driver should work with.
   - Click on the **JRE** tab and set Separate JRE to JDK 1.9 if developing for JDBC4 .3 or JDK 1.8 if developing for JDBC4.2.

4. To build the driver:

   - From the **Package Explorer**, expand the **JavaUltraLight** project.
   - Right-click on the `JavaUltraLightBuilder[43 or 42].xml` file.
   - Select **Run As**.
   - Click on **Ant Build**. This will build the Java driver using Ant and place it in the location `[INSTALL_DIRECTORY]\Examples\Source\JavaUltraLight\Lib`.

5. Open any JDBC-enabled application (e.g. SQuirreL).

   - Add the JavaUltraLight driver according to the application's instructions.
   - Add a Database/Connection using the JavaUltraLight driver according to the application's instructions. The connection URL is: `jdbc:simba://localhost`.
   - Execute a JDBC operation within the application.

If there were no problems with the example driver you built, you are now ready to set up a development project to build your own JDBC driver.

Setting Up the Project

1. Copy the `JavaUltraLight` directory and paste it to the same location. This will create a new directory called "`JavaUltraLight - Copy`". Rename the directory to something that is meaningful to you. This will be the top-level directory for your new project and DSI implementation files.

   It is very important that you take this step to create your own project directory. You might be tempted to modify the sample project files, but we strongly recommend against this, for two reasons:

   a. When you install a new release of SimbaEngine, changes you make will be lost.

   b. There may be times, for debugging purposes, that you will need to see if the same error occurs using the sample drivers. If you have modified the sample drivers, this won't be possible.

2. Rename the JavaUltraLightBuilder[43 or 42].xml file located in the `[INSTALL_DIRECTORY]\Examples\Source\<YourRenamedFolder>\Source\JavaUltraLightDSII\[JDBC_VERSION]` sub-directory. This is the `Ant builder` file (`.xml`) for your new JDBC driver. Using a text editor, open the Ant builder (.xml) and replace every instance of "JavaUltraLight" in the source code with the name of your new JDBC driver. You will also likely wish to either remove or replace the copyright information for the "doc" target. **Save** and **Close** the file.

3. Using a text editor, open the Eclipse project file (`.project`) and change the project name from "`JavaUltraLight`" to the name of your new JDBC driver.

4. Import and build the project to make sure everything compiles. At this point, the new DSII project is identical to the JavaUltraLight Driver example.

5. Searching your Java workspace for 'TODO' will find the following comments marking locations at which changes in your driver need to be made:

TODO #1: Set the driver name.	(ULDriver.java)
TODO #2: Set the driver properties.	(ULDriver.java)
TODO #3: Set the connection properties.	(ULConnection.java)
TODO #4: Set the driver-wide logging details.	(ULDriver.java)
TODO #5: Set the connection-wide logging details	(ULConnection.java)

TODO #1: Set the driver name.	(ULDriver.java)
TODO #6: Check Connection Settings.	(ULConnection.java)
TODO #7: Establish A Connection.	(ULConnection.java)
TODO #8: Create and return your Metadata Sources.	(ULDataEngine.java)
TODO #9: Prepare a Query.	(ULDataEngine.java)
TODO #10: Implement a Query Executor.	(ULQueryExecutor.java)
TODO #11: Provide parameter information. (if any)	(ULQueryExecutor.java)
TODO #12: Implement Query Execution.	(ULQueryExecutor.java)
TODO #13: Implement your Result Set.	(ULPersonTable.java)
TODO #14: Register your error messages.	(ULDriver.java)
TODO #15: Set the vendor name.	(ULDriver.java)
TODO #16: Update the component name.	(UltraLight.java)
TODO #17: Assign a unique component ID.	(UltraLight.java)
TODO #18: Set the JDBC component name.	(ULJDBC4Driver.javaULJDBC4Driver.java)
TODO #19: Set the JDBC component name.	(ULJDBC4DataSource.java ULJDBC4DataSource.java)

TODO #1: Set the driver name.	(ULDriver.java)
TODO #20: Set the subprotocol.	(ULJD4BCDriver.java ULJD4BCDriver.java)
TODO #21: Set the subprotocol.	(ULJDBC4DataSource.javaULJDBC4DataSource.java)
TODO #22: Define your custom client info properties	UL4Connection.java UL4Connection.java
TODO #23: Implements the wanted behavior	UL4Connection.javaUL4Connection

Over the next four days, you will be visiting each "TODO" and modifying the source code there.

6. As described in the previous section, use any JDBC-enabled application (e.g. SQuirreL) to test your driver.

Debugging your Driver

During the development of your driver, it may be necessary for you to trace through your driver during execution to locate problems. Most JDBC-enabled applications are written in Java themselves and they will either have a shell script/batch file to launch the application or have a configuration file where JVM options can be added.

You will need to add the following JVM options to enable debugging with the Eclipse IDE:

- `-Xdebug`
- `-Xrunjdwp:transport=dt_socket,address=localhost:8000,suspend=n,server=y`

Once you have added these options for your JDBC-enabled application, launch your application. You will now be able to attach the Eclipse debugger to the running application and debug your driver.

If you need to debug the initialization of your driver, set the suspend parameter to y. A good breakpoint to start with is inside the `ULDriver` constructor. Launch your JDBC-enabled application. The JVM will suspend execution until a debugger has been attached.

Please see the Eclipse documentation for instructions on debugging Remote Java Applications.

By the end of this day, you should have built and tested, unchanged, the example driver shipped with SimbaEngine to make sure that your installation worked properly and that your development system is properly set up. Also, you should have created, built and tested a copy of the JavaUltraLight Driver example that you will change to work with your own data store.

Day Two

Today's goal is to customize your driver, enable logging and establish a connection to your data store. To accomplish this you will visit TODO items 1 to 7.

TODO #1: Set the driver name.	(ULDriver.java)

Using your newly created project, set the constant DRIVER_NAME to the name of your driver (usually the same name you used to replace "JavaUltraLight" in steps 2 and 3 of Day One).

TODO #2: Set the driver properties.	(ULDriver.java)
TODO #3: Set the connection properties.	(ULConnection.java)

In `ULDriver’s` `setDefaultProperties()` you will set up general properties for your driver. In `ULConnection’s setDefaultProperties()`, you will set up general properties for your connection.

Some additional connection related properties required for JDBC 4 and newer also need to be handled in the following TODOs:

TODO #22: Define your custom client info properties	UL4Connection.java.java
TODO #23: Implements the wanted behaviour	UL4Connection.java UL4Connection.java

ou must set JDBC4-specific client information such as "APPLICATIONNAME", "CLIENTUSER" and "CLIENTHOSTNAME" using the `setClientInfoProperty()` method in the file(s) `UL4Connection.java` and/or `UL42Connection.java`. Load and initialize `setClientInfoProperty()` in the `loadClientInfoProperties()` method found in the same file(s).

TODO #4: Set the driver-wide logging details.	(ULDriver.java)
TODO #5: Set the connection-wide logging details	(ULConnection.java)

By default, the JavaUltraLight Driver maintains two kinds of log files: one for all driver-based calls and one for each connection created. Update these TODO’s if you do not require such fine granularity in logging.

TODO #6: Check Connection Settings.	(ULConnection.java)

Given a connection string from the JDBC-enabled application, the Simba JDBC layer will parse the connection string into key/value pairs before calling `ULConnection’s` `updateConnectionSettings()` to validate its contents. This method should validate that the entries within the `requestMap` are sufficient to create a connection. If not, you can ask for additional information from the JDBC-enabled application by specifying the additional settings in the return value.

Should any of the values received be invalid, you should throw a `BadAuthException`. In this context, 'invalid' might be text in a numeric only property, or an unknown value for a setting requiring one of several well-defined values. For the purposes of this validation, an incorrect password is not considered invalid as you are not yet trying to connect. That validation happens below in `connect()`. For your convenience, you can also use the utility functions supplied: `verifyRequiredSetting()` and `verifyOptionalSetting()`. If there are no further entries required, simply leave the responseMap empty.

TODO #7: Establish A Connection.   (ULConnection.java)

Once `ULConnection's updateConnectionSettings()` returns a `responseMap` without any required settings (if there are only optional settings, a connection can still occur), the Simba JDBC layer will call `ULConnection's connect()` passing in all the connection settings received from the application. This is where you should authenticate the user against your data store using the information provided within the `requestMap` parameter.

Should authentication fail, you should throw a `BadAuthException`. You can also use the utility functions supplied: `getRequiredSetting()` and `getOptionalSetting()`.

Should authentication succeed, you should perform any additional connection setup required by your data store.

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

Day Three

Today's goal is to return the data used to return database metadata information to the JDBC-enabled application. The majority of all JDBC-enabled applications call the following JDBC DatabaseMetaData methods:

- getCatalogs()
- getSchemas()
- getTables()
- getColumns()
- getTypeInfo()

TODO #8: Create and return your Metadata Sources.   (ULDataEngine.java)

`ULDataEngine's makeNewMetadataTable()` is responsible for creating the sources to be used to return data to the JDBC-enabled application for the various `DatabaseMetaData` methods. Each `DatabaseMetaData` method is mapped to a unique `MetadataSourceId`, which is then mapped to an underlying `IMetadataSource` that you will implement and return. Each `IMetadataSource` instance is responsible for three things:

1. Creating a data structure that holds the data relevant for your data store: `Constructor`
2. Navigating the structure on a row-by-row basis: `moveToNextRow()`
3. Retrieving data: `getMetadata`() (See Appendix A, " Data Retrieval" for a brief overview of data retrieval).

Handling Type Information Metadata

The underlying `DatabaseMetaData` method `getTypeInfo()` is handled as follows:

1. When called with `TYPE_INFO`, `ULDataEngine's makeNewMetadataTable()` will return an instance of `ULTypeInfoMetadataSource()`.
2. SimbaEngine supports all required JDBC data types. The JavaUltraLight Driver example exposes support for the following types:

BIT	CHAR	DATE
DECIMAL	DOUBLE	INTEGER
LONGVARBINARY	LONGVARCHAR	REAL
SMALLINT	TIME	TIMESTAMP
TINYINT	VARBINARY	VARCHAR
WCHAR	WLONGVARCHAR	WVARCHAR

3. For your driver, you may need to change the types returned and the parameters for the types in `ULTypeInfoMetadataSource's initializeDataTypes()`.

Handling the other Metadata Sources

The other `DatabaseMetaData` methods are handled in a similar fashion to Type Information Metadata.

1. When called with any other `MetadataSourceID`, `makeNewMetadataTable()` will return the appropriate instance of an implementation of `IMetadataSource` as illustrated by the JavaUltraLight driver. The Ultralight driver provides sample implementations of the following metadata sources which are considered the minimum requirements for a driver to provide so that an application may understand the structure of your tables and columns to generate queries:
   - TABLES – Can generate a full list of all tables in all catalogs and schemas.
   - COLUMNS – Can generate a full list of all columns in all tables. Typically, filters may be provided to restrict output to a single table.

- CATALOG_SCHEMA_ONLY – A list of all catalogs and schemas. This source and the following three are essentially subsets of the TABLES source without the table names.
- CATALOG_ONLY – Lists only the catalogs.
- SCHEMA_ONLY – Lists only the schemas (without listing which catalog they may be in).
- TABLETYPE_ONLY – Lists only types of tables. Eg. TABLE, VIEW, SYSTEM_TABLE

2. Your implementations of IMetadataSource should query your data store to obtain the appropriate metadata and provide the means to iterate through that metadata and to return the metadata.

Congratulations! You can now retrieve type metadata from within your data store. Once you have completed the work for Day Four, you will be able to retrieve the most commonly used metadata from your data store. You should be able to connect to your driver with a JDBC-enabled application and see the correct metadata returned.

Day Four

Today's goal is to enable query execution and data retrieval from within the driver. We will cover the process of preparing a query, executing the prepared query, retrieving the query result, retrieving the column information for the query result, and finally retrieving data.

Query Preparation

The first step in obtaining data from your data store is to prepare a query.

TODO #9: Prepare a Query. (ULDataEngine.java)

`ULDataEngine's prepare()` is the entry point where SimbaEngine requests queries to be prepared. You must modify this method to perform the following:

- Send a request to your data store to prepare the query.
- Handle the response from your data store.
- Create an instance of your `IQueryExecutor` implementation containing whatever information is necessary to execute the query.

If the query can be prepared, a new instance of your `IQueryExecutor` will be returned.

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

Query Execution

After a query has been prepared, a query is executed.

TODO #10: Implement a Query Executor. (ULQueryExecutor.java)

You will need to modify the constructor of ULQueryExecutor to receive information from query preparation to be used for query execution.

TODO #11: Provide parameter information. (if any) (ULQueryExecutor.java)

If your data store is capable of handling query parameters, you will need to modify the `getMetadataForParameters` method to return the relevant parameter metadata for the query and the `getNumParams` method to return the correct number of parameters for the query.

TODO #12: Implement Query Execution. (ULQueryExecutor.java)

`ULQueryExecutor's execute()` is the entry point where SimbaEngine requests queries to be executed. You must modify this method to perform the following:

- Serialize all input parameters (if any) in a form that can be consumed by the data store.
- Send a request to your data store to execute the query.
- Retrieve all output parameters (if any) from the data store.
- Retrieve query results from the data store.

Query Results

After a query has been executed, the query results are returned in an implementation of the `IResultSet` interface. The `DSISimpleResultSet` class provides a partial implementation of the interface to simplify the task of implementing a basic forward-only, read-only result set.

TODO #13: Implement your Result Set. (ULPersonTable.java)

`ULPersonTable` implements a simple in-memory table. In general, your “table” class can represent the results of a query that may involve more than a single table but for simplicity, this tutorial assumes a query involving a single table.

The next sections describe the changes you must make to `ULPersonTable` for it to work with your data store.

- Return the columns defined for your table.
    - `initializeColumns()`: This method must be modified so that, for each column defined in the query, you define the `ColumnMetadata` in terms of SQL types.

Here is an example of code for the new method:

```
Get all the column information from your data store for the table
For Each Defined Column
{
 // Set the argument of the following method call to the SQL Type that
 // maps to the data store type of the column.
 TypeMetadata typeMetadata =
  TypeMetadata.createTypeMetadata(Types.VARCHAR);

 // Depending on SQL type, set different properties:
 if (character type)
 {
  typeMetadata.setIntervalPrecision(m_settings.m_maxColumnSize);
 }
 else if (exact numeric type)
 {
  typeMetadata.setScale(scale);
 }

 // Create the column metadata.
 ColumnMetadata columnMetadata = new ColumnMetadata(typeMetadata);
 columnMetadata.setCatalogName(m_catalogName);
 columnMetadata.setSchemaName(m_schemaName);
 columnMetadata.setTableName(m_tableName);
 columnMetadata.setName("column name");
 columnMetadata.setLabel("localized column name");
 columnMetadata.setNullable(Nullable.NULLABLE);

 if ( character type )
 {
  columnMetadata.setColumnLength(m_settings.m_maxColumnSize);
 }

 // Add the column metadata to the list of column metadata.
 m_columns.add(columnMetadata);
}
```

- Data Retrieval
  - `doMoveToNextRow()`
  - `getData()`

These methods are responsible for navigating a data structure containing information about one table in your data store, and retrieving data from that table.

It is best to implement a class that provides a streaming interface for the data in the table within your data store. It should also provide the ability to navigate forward from one table row to the next. The class should be able to navigate across columns within the row and to read the data associated with the current row and column combination.

In the JavaUltraLight Driver, `ULPersonTable` stores its data in an in-memory Java object list. Each object represents a row of data and each member variable in the object represents a column of data. Its `getData` method takes a `column index` and uses it to determine from which member variable of the current row/object to retrieve data. See Appendix A, "Data Retrieval", for a brief overview of data retrieval.

- `doCloseCursor()`

This is a callback method called by SimbaEngine to indicate that data retrieval has completed and that you may now perform any tasks related to closing any associated result set in your data store.

- `hasMoreRows()`

This method should indicate whether there are more rows to fetch after the current row.

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

Day Five

Today's goal is to start productizing your driver. These steps, similar to day one and two are to finish making the driver uniquely yours by changing some additional references to the sample to your driver.

TODO #14: Register your error messages. (ULDriver.java)

For the purpose of prototyping, this TODO is purely informational. By default, the driver's messages.properties file resides in the same package as the ULDriver class. You can modify the code to look in a different package location for the messages file or to customize the name of the file.

TODO #15: Set the vendor name.	(ULDriver.java)

All error messages returned by the driver begin with the vendor name. The default vendor name is "Simba". Simply uncomment the call to the `setVendorName` method and replace "`vendorName`" with an appropriate name for your organization. This will rebrand your converted JavaUltraLight driver for your organization.

TODO #16: Update the component name.	(UltraLight.java)

All error messages returned by your DSII contain the component name. Simply change "`UltraLightDSII`" to a name relating to your driver. This will rebrand your converted JavaUltraLight Driver for your organization.

TODO #17: Assign a unique component ID.	(UltraLight.java)

For the purpose of prototyping, this TODO is purely informational. The default component ID is usually sufficient for most drivers. The component ID is used to identify the component from which an error has been generated so that the correct component name can be included in the error message.

TODO #18: Set the JDBC component name.	(ULJDBC4Driver.java ULJDBC4Driver.java)
TODO #19: Set the JDBC component name.	(ULJDBC4DataSource.java ULJDBC4DataSource.java)

For the purpose of prototyping, this TODO is purely information. By default, the JDBC component name is "JDBC Driver". You can change this name if you would like a different name to be used when errors are generated in the JDBC layer.

TODO #20: Set the subprotocol.	(ULJDBC4Driver.java ULJDBC4Driver.java)
TODO #21: Set the subprotocol.	(ULJDBC4DataSource.jav ULJDBC4DataSource.java)

By default, the subprotocol to which the driver will respond is "`simba`". Simply change "`simba`" to a name relating to your driver. This will rebrand your converted JavaUltraLight driver for your organization.

Remaining Productization

Up to this point, the package and class names have not been changed to make it easier to follow the tutorial as changes were made to the driver.

To complete productization, rename/refactor all packages, files, and classes by changing all instances of the following items:

- The word "`simba`" in package names to an appropriate name for your organization.
- The word "`UltraLight`" to an appropriate name for your driver, usually related to the name chosen in steps 2 and 3 on Day One.
- The letters "`UL`" to a two-letter abbreviation of your choice.

You now have a driver that can be used by JDBC-enabled applications to query and retrieve data from your data store, and that has been productized for your organization.

Appendix A: Data Retrieval

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

1. Each `IMetadataSource` implementation of `getMetadata()`
2. `ULPersonTable’s getData()`

Both methods will provide a way to uniquely identify a column within the current row. For `IMetadataSource`, SimbaEngine will pass in a unique column tag (see `MetadataSourceColumnTag`). For `ULPersonTable`, SimbaEngine will pass in the column index.

In addition, both methods accept the following three parameters:

1. `data`

   The `DataWrapper` into which you must copy your cell’s value. This class is a wrapper around an Object managed by SimbaEngine. You simply call its `set<Data Type>()` and `get<Data Type>()` methods to store and access the data according to the `java.sql.Type`. The data you set must be represented as the Object or primitive data type that is accepted by the set methods for that java.sql.Type. If your data is not stored as the appropriate type, you will need to write code to convert from your native format.

   The type of this parameter is governed by the metadata for the column that is returned by the class. Thus, if you create the `TypeMetadata` of column 1 in `ULPersonTable’s initializeColumns()` as `Types.INTEGER`, then when `ULPersonTable’s getData()` is called for column 1, you will be passed a `DataWrapper` that wraps a `Long` data type. For `IMetadataSource`, the type is associated with the column tag (see `MetadataSourceColumnTag`).

2. `offset`

   Some data types can be retrieved in parts. This value specifies where in the current column the value should be copied from. The value is usually 0.

3. `maxSize`

   The maximum size (in bytes) that can be copied into the type. For character or binary data, copying data over this amount can result in a data truncation warning, or worse, a heap-violation.

Third Party Licenses

4. "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)"
5. The word 'cryptographic' can be left out if the rouines from the library being used are not cryptographic related :-).
6. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement:

"This product includes software written by Tim Hudson (tjh@cryptsoft.com)"

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

The licence and distribution terms for any publically available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution licence [including the GNU Public Licence.]

Expat License

"Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd

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

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

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

Stringencoders License

Copyright 2005, 2006, 2007

Nick Galbreath -- nickg [at] modp [dot] com

All rights reserved.

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