Magnitude Simba SDK

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

Version 10.2.2

October 2022

Copyright

Contact Us

Magnitude Software, Inc.

www.magnitude.com

Table of Contents

About this Guide

Purpose

This guide explains how to use the Magnitude 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:**
>
> 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 and driver managers in connecting to a data store.
- Exposure to SQL.

Variables Used in this Document

The following variables are used in this document:

Variable	Description
[INSTALL_DIR]	Installation directory for the SimbaEngine X SDK. **Default value on Windows platforms**: `C:\Simba Technologies\SimbaEngineSDK\10.2` **Default value on Linux, Unix, and macOS platforms**: *[UNTAR_DIR]*`/SimbaEngineSDK/10.2`
[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`.

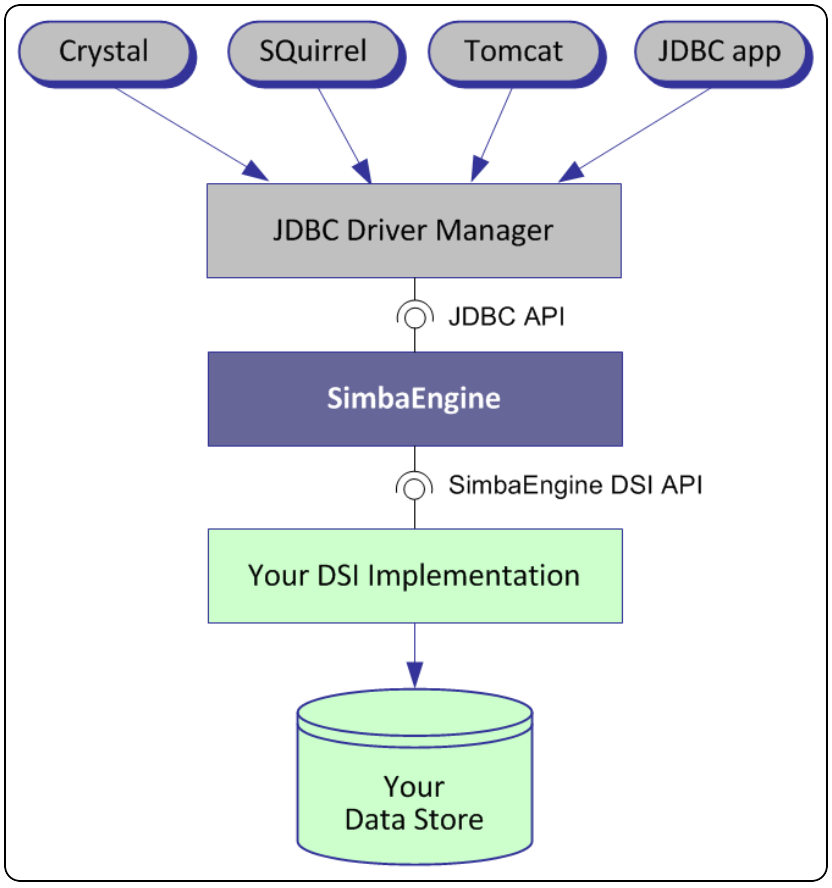
SimbaEngine X SDK Overview

Java applications typically use JDBC drivers to connect to data stores from which they read and write data. These applications support the JDBC protocol to enable connection with any driver that also supports JDBC. A driver exposes the JDBC protocol to the application and another API, such as SQL or a custom API, to the data store.

The SimbaEngine X SDK Solution

When you base your driver on the SimbaEngine X SDK, you leverage its error checking, session management, data conversion, optimization, and other low-level implementation details. The SimbaEngine X SDK uses JDBC to communicate with the driver manager, and a simple API (called the Data Store Interface API or DSI API) to communicate with the data store. The DSI API defines the primitive operations needed to access a data store.

The figure below shows a typical JDBC stack:



SDK developers create an implementation of a DSI (also known as a DSI Implementation or DSII) that applications use to access the particular data store in the process of executing an SQL statement. In the final executable, the components from SimbaEngine X SDK take responsibility for meeting the data access standards, while the custom DSI implementation takes responsibility for accessing the data store and translating it to the DSI API.

JDBC applications use this executable when connecting to the data store in the process of executing an SQL statement.

About the JavaUltraLight Sample Driver

This guide explains how to make the following modifications to the JavaUltraLight sample driver:

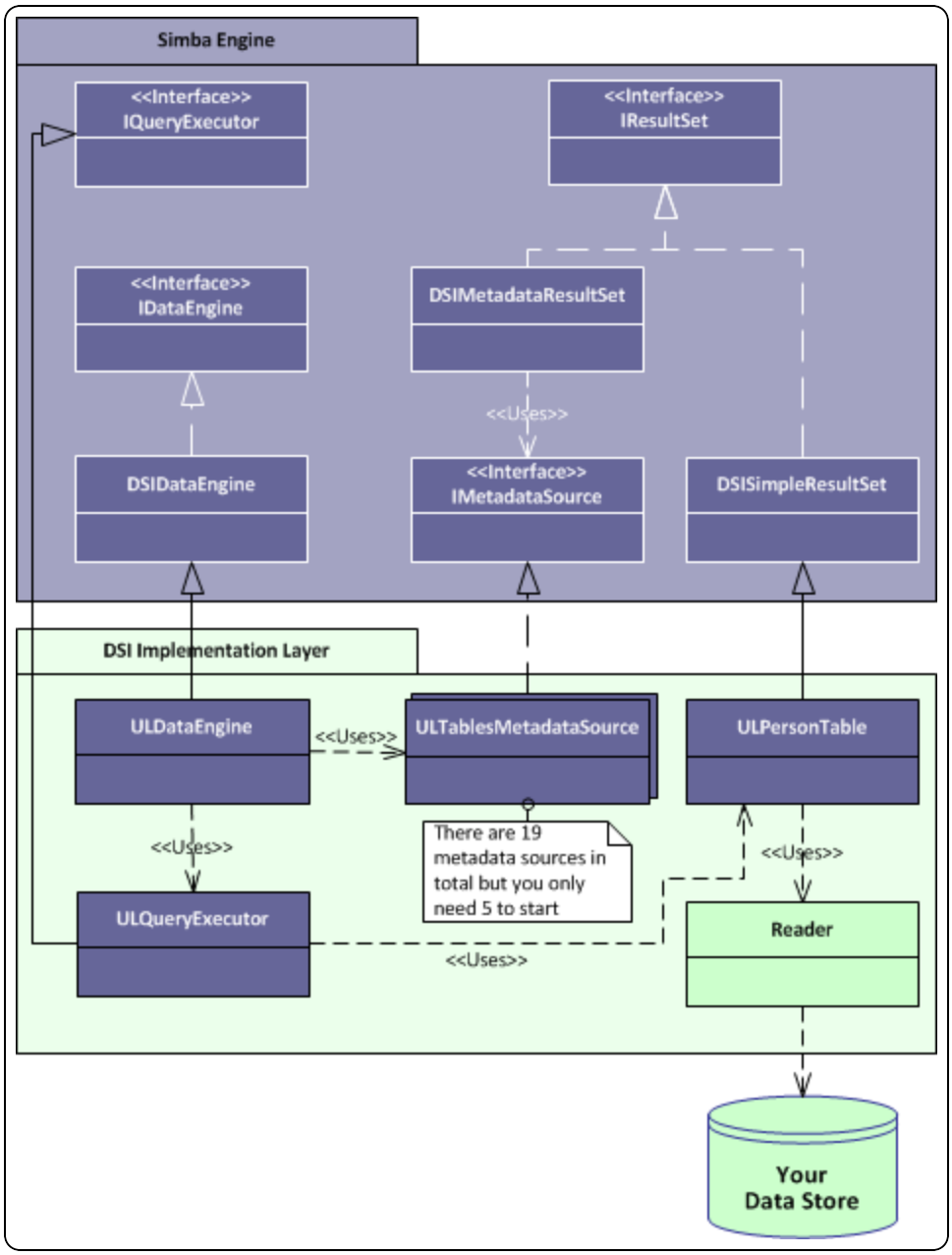
- Connect to your data store instead of the sample data store.
- Retrieve the data and metadata from your data store.
- Prepare and execute queries.
- Rebrand the contents of log files, error messages, and driver configuration properties.

Using the JavaUltraLight sample driver to prototype a DSI implementation for a custom data store helps you understand how the SimbaEngine X SDK works. By removing the shortcuts and simplifications implemented in the JavaUltraLight driver, you can use it as the foundation for a commercial DSI implementation.

Sample JDBC Driver Architecture

This section provides a technical overview of the sample JDBC driver that is provided with the SimbaEngine X SDK.

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.



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. 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

Day One

The Day One instructions explain how to install the Simba SDK, compile the sample JDBC connector, and review the configuration information created at compile time.

After the sample JDBC connector is successfully compiled, it is used to retrieve data from the data source that is included with the Simba SDK. The sample JDBC connector is then used to create the framework for a custom JDBC connector, which is renamed and used to retrieve sample data.

At the end of the day, you will have compiled, built and tested your custom JDBC connector.

Install the SimbaEngine X SDK

The SimbaEngine X SDK contains a sample JDBC driver, JavaUltraLight, that can be used to create your custom JDBC driver. The SimbaEngine X SDK is supported on Windows, Linux, Unix, or macOS platforms.

Installing on Windows

To install SimbaEngine X SDK on Windows:

1. If a version of the SimbaEngine X SDK is installed on your machine, uninstall it before installing the new version.
2. Run the SimbaEngine X SDK setup executable and follow the installation instructions. The following environment variables are set when the SimbaEngine X SDK is installed:
   - **SIMBAENGINE_THIRDPARTY_DIR** - [`INSTALL_DIR]\SimbaEngineSDK\10.2\DataAccessComponents\ThirdParty`
   - **SIMBAENGINE_DIR** - [`INSTALL_DIR]\SimbaEngineSDK\10.2\DataAccessComponents`

> **Note:**
>
> The SimbaEngine X SDK installer defines environment variables for the user who ran the installation. If you log in to Windows as a different user, you will not be able to run the driver.

You are now ready to build and test the sample driver.

Installing on Linux

On Linux platforms, SimbaEngine X SDK is distributed as `SimbaEngineSDK [BUILD].tar.gz`, where *[BUILD]* is the build number and platform. This file is a TAR archive that has been compressed with gzip.

To install SimbaEngine X SDK on Linux:

1. Open a command prompt and change to a directory where you would like to install the SimbaEngine X SDK.
2. Uncompress `SimbaEngineSDK[BUILD].tar.gz`. Enter

   ```
   gzip -d SimbaEngineSDK[BUILD].tar.gz
   ```

   Where *[BUILD]* is the build number.
3. Install the SimbaEngine X SDK. Enter the following:

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

You are now ready to build and test the sample driver.

Install the Evaluation License on page 37

Build the Sample JDBC Driver

These instructions assume you will be working with the Eclipse integrated development environment (IDE).

Note:

- Ensure that you have the environment variable JAVA_HOME pointing to the root of your JDK.
- If you make changes to Windows environment variables, you must restart Eclipse so it will detect the changes.

To build the sample driver:

1. Ensure that you have the Windows environment variable **JAVA_HOME** pointing to the root of your JDK.
2. Ensure that you have the correct JDK version installed on your machine, depending on the version of JDBC that your custom driver will support. Each version of JDBC requires a given minimum version of the JDK:

JDBC version	Minimum required JDK version
JDBC4.2	JDK 1.8
JDBC4.3	JDK 9

Note:

You can build JDBC4.2 or JDBC4.3 with a newer JDK version, as long as it supports the -release option for 1.8 or 9 when invoking javac.

3. Use Eclipse to import the JavaUltraLight project as an existing project into your workspace.

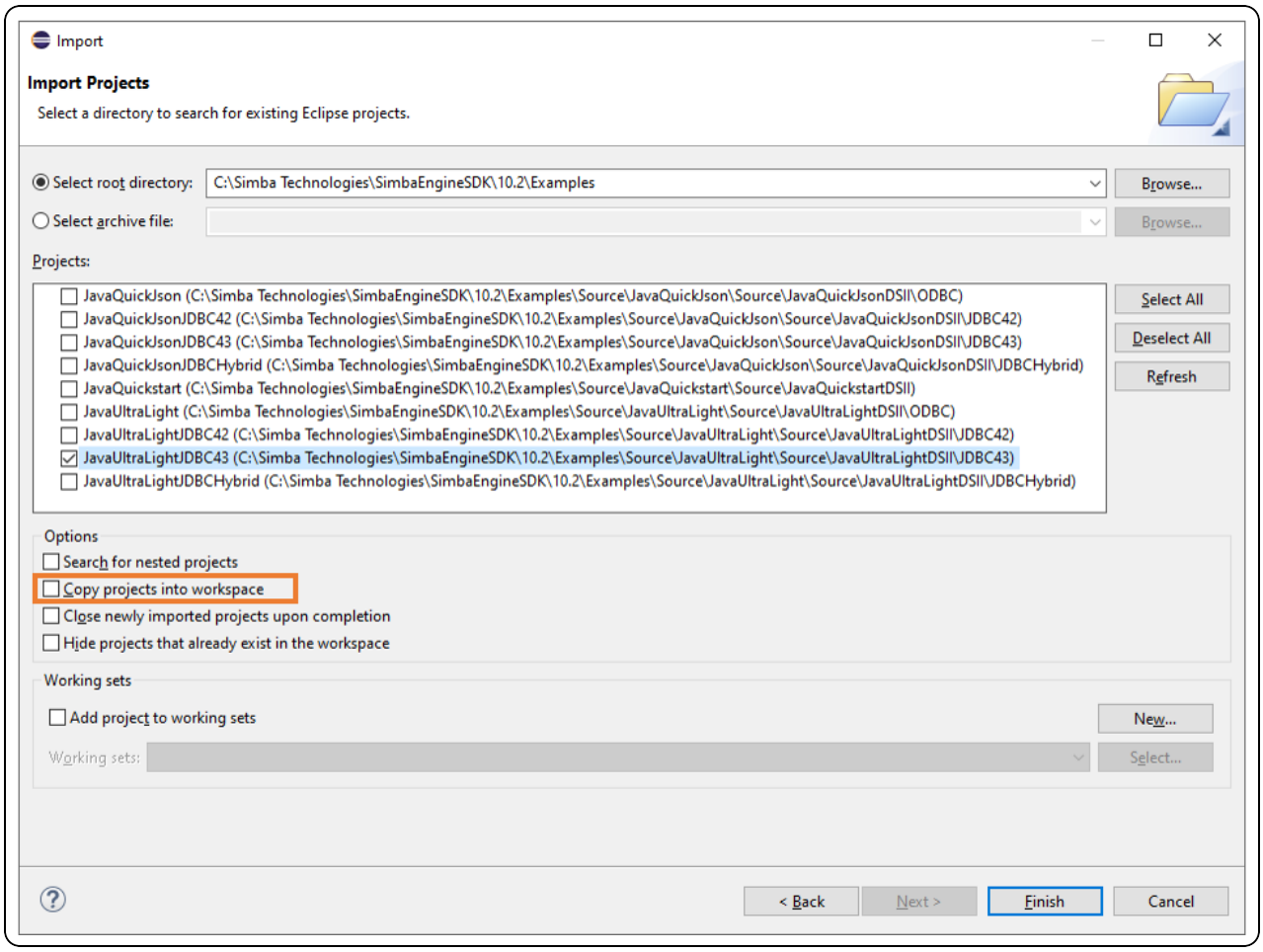
   The project files are located under *[INSTALL_DIR]*\Examples\Source\JavaUltraLight\Source\JavaUltraLightDSII*[JDBC_VERSION]*, where *[JDBC_VERSION]* is the version of JDBC that the sample driver supports. The hybrid project files are located under *[INSTALL_DIR]*\Examples\Source\JavaUltraLight\Source\JavaUltraLightDSII*JDBCHybrid*.

   Important:

   Uncheck the box **Copy projects into workspace**.

   If this box is checked, the project is copied and the build will not succeed due to relative paths in the build files. To put the project in a different location, copy the entire project structure.

   For example, select the **JavaUltraLightJDBC43** project to build a sample Java driver that supports JDBC 4.3:



4. Update the classpath.
    a. From the Main Menu, select **Project>Properties>Java Build Path>Libraries**.
    b. Select **SIMBAENGINE_DIR ->Edit ->Variable -> New**.
    c. Create a new classpath variable called SIMBAENGINE_DIR with the value `[INSTALL_DIR]\DataAccessComponents`, for example: `C:\Simba Technologies\SimbaEngineSDK\10.2\DataAccessComponents`.
    d. Select **OK** and perform a full rebuild of the workspace when prompted. When the workspace build is successful, you are ready to build the driver.
5. Configure the build for your chosen version of JDBC:
    a. From the Package Explorer, expand the **JavaUltraLight** project.
    b. Right-click on the **JavaUltraLightBuilder*[JDBC_VERSION]*.xml** file.
    c. Select **Run As** and select the second **Ant Build** (at the bottom of the list).

d. On the Targets tab select **debug-*[Java_Version]***, where *[Java_Version]* is 8 for JDBC 4.2, 9 for JDBC 4.3, and hybrid for a hybrid driver.
e. Click on the **JRE** tab and set **Separate JRE** to one of the following minimum JDK versions:
   - If developing for JDBC4.2 or hybrid, select JDK 1.8 or later.
   - Or, if developing for JDBC4.3, select JDK 9 or later.
f. Click **close**.

6. Build the driver using the following steps:
   a. Right-click on the **JavaUltraLightBuilder*[JDBC_VERSION]*.xml** in the Package Explorer.
   b. Select **Run As** and select the first **Ant Build** (at the top of the list).

      This will build the Java driver using Ant and place it in the location `[INSTALL_DIR]\Examples\Source\JavaUltraLight\Lib`.

You have built the sample JDBC driver.

Test the Sample JDBC Driver

Use any JDBC-enabled application to test the JavaUltraLight driver. In this example, we use DbVisualizer, a free JDBC test application.

To test the sample driver:

1. Open DbVisualizer.
2. Add the JavaUltraLight driver that you built in Build the Sample JDBC Driver on page 13:
   a. Select **Tools** > **Driver Manager**.
   b. Select the "+" icon to add a new driver.
   c. In the pop-up, select **Custom** as the template for your driver.
   d. In Driver Settings, enter a name for your driver. You can choose any name.
   e. Under **Driver artifacts and jar files**, select the "+" icon and select **Add files**.
   f. Navigate to the JAR file that you built in Build the Sample JDBC Driver on page 13. For example: `[INSTALL_DIR]\Examples\Source\JavaUltraLight\Lib\JavaUltraLightJDBC42.jar`. Then, click **Open**.
   g. Close the Driver Manager window.

   The JavaUltraLight driver is added to the DbVisualizer application.

3. Add a connection to the driver:
   a. Select **Database** > **Create Database Connection** and choose the driver you added in the previous step.
   b. Enter the name of the connection (you can enter any name), then click **Next**.
   c. Under the **Database** section, select **Settings Format** and set it to **Database URL**.
   d. Enter the following string for the Database URL:

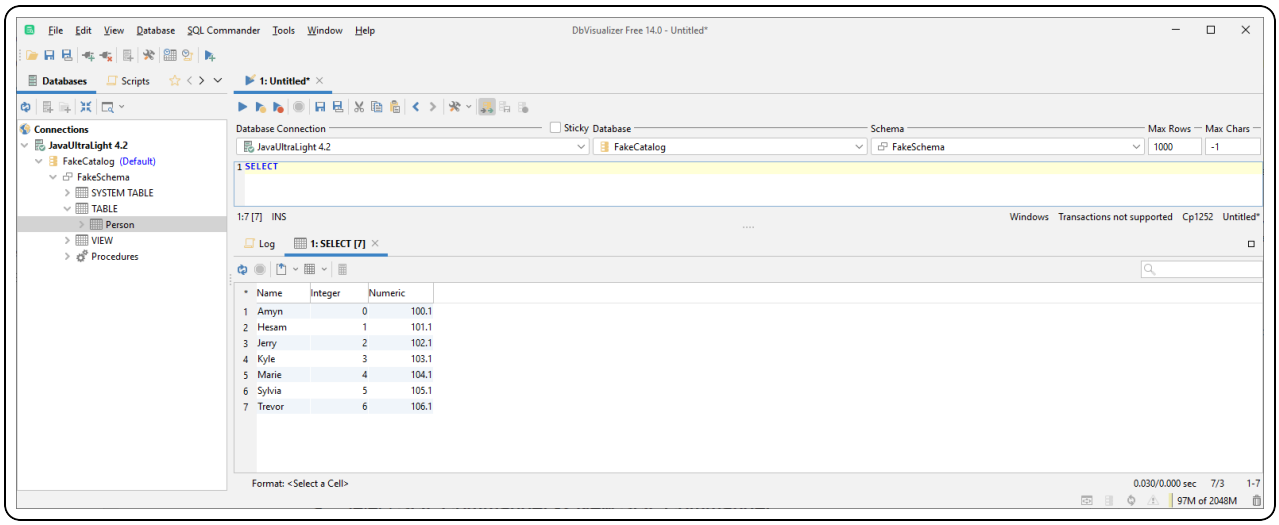
      ```
      jdbc:simba://localhost
      ```

      Or, `jdbc:simba://localhost;PWD=a;UID=b;`

      You can view the required settings of the sample driver by looking in QJPropertyKey.java or ULPropertyKey.java. For information on using required and optional settings, see Establish a Connection on page 24.
   e. Click the **Connect** button.

   The connection is created and the database tables are listed in the Connections panel.
4. Use the connection to execute an SQL command, for example SELECT.
   a. Select **SQL Commander** -> **New SQL Commander**.
   b. From the Database Connection drop list, select the connection you added in the previous step.
   c. Enter the SQL command, for example, “SELECT”.
   d. Click the blue arrow to execute the statement.

   The results are displayed. For example:



Once the JavaUltraLight sample driver successfully executes an SQL statement and retrieves the results, you are ready to build your own custom JDBC driver.

Set up the Custom JDBC Driver Project

We recommend that you copy the sample project as a starting point for your own driver project.

> **Important:**
>
> It is very important that you create your own project directory. You might be tempted to simply modify the sample project files, but we strongly recommend that you create your own project directory. If you simply modify the sample project files:
>
> - All your changes will be lost when you install a new version of the SDK.
> - You will lose your frame of reference for debugging.
>   There may be times, for debugging purposes, that you will need to see if the same error occurs using the sample connectors. If you have modified the sample connectors, this won't be possible.

To set up your own driver project:

1. In Windows Explorer, copy the `JavaUltraLight` folder and paste it to the same location. This will create a new directory called "JavaUltraLight - Copy". Rename the directory to reflect the name of the driver you will create. This will be the top-level directory for your new project and DSI implementation files.

2. Rename the Ant build file for your own driver project.
   a. Open the Ant build file:

      `[INSTALL_DIR]\Examples\Source\<YourRenamedFolder>\Source\JavaUltraLightDSII\[JDBC_VERSION]\JavaUltraLightBuilder[JDBC_VERSION].xml`

   b. Using a text editor, replace every instance of "JavaUltraLight" in the XML text with the name of your new JDBC driver.
   c. (Optional). Remove or replace the copyright information for the **doc** target.
   d. Save and close the file.
3. Rename the Eclipse project.
   a. Open the Eclipse project file (`.project`) in a text editor.
   b. Change the project name from "JavaUltraLightJDBC*[JDBC_VERSION]*" to the name of your new JDBC driver.
   c. Save and close the `.project` file.
4. Import and build the project to ensure it compiles without errors.
5. View the TODO items.

   You can get an overview of what changes you will be making to the driver code by looking at the TODO tasks in the project:

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 #6: Check Connection Settings.	(ULConnection.java)
TODO #7: Establish a connection to your data store.	(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 for handling by DSIMessageSource.	(ULDriver.java)
TODO #15: Set the vendor name for the error messages.	(ULDriver.java)
TODO #16: Update the component name.	(UltraLight.java)
TODO #17: Assign a unique component ID value to the messages.	(UltraLight.java)
TODO #18: Set the JDBC Component Name for error messages originating from the JDBC layer.	(ULJDBC4Driver.java, ULJDBC41Driver.java, ULJDBC42Driver.java, or ULJDBCHybridDriver.java)
TODO #19 : Set the JDBC Component Name for error messages originating from the JDBC layer.	(ULJDBC4DataSource.java, ULJDBC41DataSource.java, ULJDBC42DataSource.java, or UL JDBCHybridDataSource.java)
TODO #20: Set the subprotocol to which this driver will respond.	(ULJD4BCDriver.java, ULJD41BCDriver.java, ULJD42BCDriver.java, or ULJDBCHybridDriver.java)
TODO #21: Set the subprotocol to which this driver will respond.	(ULJDBC4DataSource.java, ULJDBC41DataSource.java, ULJDBC42DataSource.java, or UL JDBCHybridDataSource.java)
TODO #22: Define your custom client info properties.	UL4Connection.java, UL41Connection.java, and/or UL42Connection.java
TODO #23: Implement the wanted behaviour.	UL4Connection.java, UL41Connection, and/or UL42Connection.java

6. Use any JDBC-enabled application to test your custom JDBC driver, as described in Test the Sample JDBC Driver on page 16.

Debug the Custom JDBC Driver

During the development of your driver, you may want to trace through your driver during execution. 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.

Modify the JDBC Application to Enable Debugging

You can run your driver in the Eclipse development environment in debug mode. To enable your JDBC application to work with Eclipse in debug mode, add the following JVM options to your JDBC application:

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

Once you have added these options, launch your JDBC application. You will now be able to attach the Eclipse debugger to the running application and debug your driver. Launch your JDBC-enabled application, and the JVM will suspend execution until a debugger has been attached.

For more information, see the Eclipse documentation for instructions on debugging Remote Java Applications.

Debugging the Driver Initialization

If you need to debug the initialization of your driver, set the `suspend` parameter to `y`. The JVM will suspend execution until you attach a debugger.

A good place to put a breakpoint is the `ULDriver` constructor. Once this constructor is called, you know that:

- DSII has been successfully built, located, loaded, and instantiated by the driver manager.
- Your concrete driver implementation is being invoked.

Example: Debugging with DBVisualizer and Eclipse

You can use Eclipse to run your sample JDBC driver in debug mode, then run a query from DBVisualizer. This allows you to step through the code in your custom JDBC driver. Ensure you have configured a connection to your driver as described in Test the Sample JDBC Driver on page 16.

1. Set a breakpoint in your project.
   a. In Eclipse, open the file `com.simba.ultralight.dataengine.ULPersonTable.java`.
   b. Set a breakpoint in the constructor.

2. Configure DBVisualizer:
   a. In DBVisualizer, select **Tools** > **Tool Properties**.
   b. In the Java VM Properties window, specify the following overridden Java VM properties:
   ```
   -Xdebug
   -Xrunjdwp:transport=dt_
   socket,address=localhost:8000,suspend=n,server=y
   ```
   c. Click **Apply** and **OK**.
3. Configure a debug target in Eclipse:
   a. Select **Run** > **Debug Configurations**.
   b. Create a Remote Java Application configuration for your driver project. Ensure the Host and Port match the Host and Port specified in DBVisualizer (*localhost* and *8000*).
   c. Select **Debug**. Note: if the connection is refused, restart DBVisualizer.
4. In DBVisualizer, execute an SQL command against the driver, as explained in Test the Sample JDBC Driver on page 16.

The breakpoint is hit in your custom driver project, and Eclipse opens in the debug perspective.

Summary of Day One

You have successfully completed the following tasks:

- Built and tested the JavaUltraLight sample connector.
  This verifies that your installation and development environment are properly configured.
- Created, built, and tested a custom connector project by copying the JavaUltraLight connector.
  You can use this project as a framework to create your custom JDBC connector.

Day Two

Day Two instructions explain how to customize your JDBC connector, enable logging, and establish a connection to your data store.

Set Driver Properties

Set the properties for your custom JDBC driver.

Set the driver name

Using your custom project, set the constant DRIVER_NAME to the name of your driver (usually the same name you used to replace “JavaUltraLight” in Day One). This is used to name your driver, and will appear in the JDBC driver manager.

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

Set the driver properties

In `ULDriver's setDefaultProperties()` you can set up general properties for your driver. Use this method to override any of the default values that are set by `DSIDriver`.

TODO #2: Set the driver properties. (QJDriver.java)

Set the connection properties

In `ULConnection's setDefaultProperties()`, set up general properties for your connection. Use this method to override any of the default values that are set by `DSIConnection`.

TODO #3: Set the connection properties. (QJConnection.java)

Set the custom client properties

When building your driver, you must set JDBC-specific client information such as “APPLICATIONNAME”, “CLIENTUSER” and “CLIENTHOSTNAME” using the `setClientInfoProperty()` method in the following file:

- QJConnection.java

Load and initialize `setClientInfoProperty()` in the `loadClientInfoProperties()` method found in the same file(s).

TODO #22: Define your custom (ULJDBC42DataSource.java,

client info properties.	ULJDBC43DataSource.java, or ULJDBCHybridDataSource.java)
TODO #23: Implement the wanted behaviour.	(QJConnection.java)

Set Logging Details

You can set the driver-wide and the connection-wide logging.

In these TODOs, a `DSILogger` object is created to handle logging. A unique file name is passed into the `DSILogger`, specifying the name of the log file.For the connection-wide logging, the string "_conn" and the connection ID are also passed in so that one log is created for each connection.

If you do not require such fine granularity in logging, you can modify these TODO's to use the same file name for both the driver-wide and the connection-wide logging.

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

Establish a Connection

To establish a connection to the data store, verify the connection settings then authenticate the user.

Check Connection Settings

When the Simba JDBC layer is given a connection string from a JDBC-enabled application, the Simba JDBC layer parses the connection string into key-value pairs. Then, the entries in the connection string and the DSN are put into a `requestMap` object and passed to `QSConnection::UpdateConnectionSettings()` for validation.

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

`UpdateConnectionSettings()` receives all the incoming connection settings that are specified in the DSN that was used to establish the connection. Modify this method to validate that the entries in the `requestMap` are sufficient to create a connection:

- Use `verifyRequiredSetting()` for required settings (settings that must be included in order to establish a connection).
- Use `verifyOptionalSetting()` for optional settings.

If all of the required settings do not exist, you can ask for additional information from the JDBC-enabled application by specifying the additional settings in the `responseMap` output parameter.

The sample JDBC driver uses `verifyRequiredSetting()` and `verifyOptionalSetting()` to perform the validation, and uses the `requestMap` to populate the `responseMap`. Your custom JDBC driver can also use these functions.

If any of the values are invalid, throw a `BadAuthException`. If no further entries are required, leave the `responseMap` empty.

Establish a Connection

Once `ULConnection.updateConnectionSettings()` returns a `responseMap` without any required settings (if there are only optional settings, a connection can still occur), the SimbaEngine X SDK JDBC layer will call `ULConnection.connect()`, passing in all the connection settings received from the application.

TODO #7: Establish a Connection to your data store. (ULConnection.java)

At this point, authenticate the user against your data store using the information provided in the `requestMap` parameter. For example, you could extract the username and password key/value pairs and then authenticate them against the underlying data store.

Note:

The JavaUltraLight driver does not actually establish a connection a connection to a real data store. All data in this sample JDBC driver is stored in-memory.

You can use the utility functions `getRequiredSetting()` and `getOptionalSetting()` to extract the settings from the `requestMap`. If authentication fails, throw a `BadAuthException`. If authentication succeeds, perform any additional connection setup required by your data store.

Summary of Day Two

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

Day Three

Day Three instructions explain how to return the data used to provide database metadata information to the JDBC-enabled application.

Create and Return Metadata Sources

Create and return your Metadata Sources. Metadata Sources correspond to the result sets that are returned when calling a Catalog function, for example, SQLTables or SQLColumns.

Most JDBC applications require a driver to support the following JDBC `DatabaseMetaData` methods, at a minimum:

- 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 each of the `DatabaseMetaData` methods. Each `DatabaseMetaData` method is mapped to a unique `MetadataSourceId`, which is then mapped to an underlying `IMetadataSource` that you 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 Data Retrieval on page 36 for a brief overview of data retrieval.

Handle Type Information Metadata

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

1. When called with `TYPE_INFO`, `ULDataEngine.makeNewMetadataSource()` will return an instance of `ULTypeInfoMetadataSource()`.

2. The SimbaEngine X SDK supports all required JDBC data types. The JavaUltraLight sample driver 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()`.

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

Summary of Day Three

Your custom JDBC connector can now return type metadata. You can use a JDBC-enabled application to connect to your connector and retrieve type metadata from within your data store

Day Four

Day Four instructions explain how to enable data retrieval from within the connector.

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 the SimbaEngine X SDK requests the preparation of a query. 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.
- Implement an instance of `IQueryExecutor`, containing the information required to execute the query.

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

Your implementation of `IQueryExecutor` should wrap your statement context to the prepared query. Query preparation consists of three main steps:

1. Generate and send the request to your data source for query preparation.
2. Handle the response.

   For each statement in the query, retrieve its column metadata information prior to query execution. Derive from `DSISimpleResultSet` to create your representation of a result set. See `ULPersonTable`.
3. Create an instance of `IQueryExector`, seeding it with the results of the query. See `ULQueryExecutor`.

Query Execution

After a query has been prepared, it is executed.

Implement a Query Executor

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

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

Return Parameter Metadata

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

Implement Query Execution

`ULQueryExecutor::execute()` is the entry point where the SimbaEngine X SDK requests that queries be executed. Modify this method to perform the following:

- Serialize all input parameters (if any) in a form that can be consumed by the data store. If your data store does not support parameter streaming for DATA_AT_EXEC parameters, then you need to re-assemble them from your parameter cache. See `pushParamData()`.
- Use the `Execute()` message to send a request to your data store to execute the query.
- Retrieve all output parameters (if any) from the data store, then and update the contexts with their contents.

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

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 for a SELECT query.

```
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.

Changes to the ULPersonTable class

This section explains the changes you must make to `ULPersonTable` for it to work with your data store.

Return the columns defined for your table

Modify `initializeColumns()`. For each column defined in the query, define the `ColumnMetadata` in terms of SQL types.

Example: pseudocode for your new initializeColumns() 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);
}
```

Implement Data Retrieval

The `doMoveToNextRow()` and `getData()` 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 Data Retrieval on page 36 for a brief overview of data retrieval.

The `doCloseCursor()` method is a callback method called by the SimbaEngine X SDK 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.

The `hasMoreRows()` method should indicate whether there are more rows to fetch after the current row.

Summary of Day Four

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

Day Five

Day Five instructions explain how to rebrand your custom JDBC connector.

Rebrand the Custom JDBC Driver

By default, the sample JDBC driver uses the brand "Simba" in error messages, class names, and other areas. You can rebrand your custom JDBC driver by changing this brand to reflect your own company or product.

Register the Error Messages

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 #14: Register your error messages for handling by DSIMessageSource.	(ULDriver.java)

Set the Vendor Name

All error messages returned by the driver begin with the vendor name, by default "Simba". To rebrand the vendor name for your custom JDBC driver, uncomment the call to `setVendorName()` and replace `vendorName` with your custom name.

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

Update the Component Name

A DSII contains a unique component name that is used to identify the driver during logging. All error messages returned by the DSII contain the DSII's component name. In your custom JDBC driver, change the default name "JavaUltraLightDSII" to a custom name. This rebranding identifies messages that are logged by the custom DSII.

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

Assign a Component ID

This step is optional, because the default component ID is usually sufficient for most drivers. The packages and classes that make up a driver each have a unique component ID that is added to the logging messages. This makes it easy to identify which component logged a message in the log files.

TODO #17: Assign a unique component ID value to the messages.	(UltraLight.java)

Assign a Component Name

The default JDBC component name is “JDBC Driver”. The component name is included when errors are generated in the JDBC layer. The component name can be customized.

TODO #18 : Set the JDBC Component Name for error messages originating from the JDBC layer.	(ULJDBC42Driver.java, JDBC43Driver.java, or ULJDBCHybridDriver.java)
TODO #19: Set the JDBC Component Name for error messages originating from the JDBC layer.	(ULJDBC42DataSource.java ULJDBC43DataSource.java, or UL JDBCHybridDataSource.java)

Set the Subprotocol

A protocol and sub protocol act as a URL for the driver and are required in order for the JDBC driver manager to locate, expose, and use the driver. By default, the driver responds to the subprotocol “`simba`”. To rebrand your custom JDBC driver, change “`simba`” to a custom subprotocol.

TODO #20: Set the subprotocol to which this driver will respond.	(ULJDBC42Driver.java, ULJDBC43Driver.java, or ULJDBCHybridDriver.java)
TODO #21: Set the subprotocol to which this driver will respond.	(ULJDBC42DataSource.java, ULJDBC43DataSource.java, or UL JDBCHybridDataSource.java)

Remaining Productization

When writing a custom JDBC driver, the final step in productization is to refactor and rename all the packages, files, and classes. Make the following changes:

- The word “simba” in package names to reflect your organization.
- The word “UltraLight” to reflect your driver name. This is usually related to the name chosen in steps 2 and 3 on Day One.
- The letters “UL” to a custom two-letter abbreviation.

Conclusion

You have written a custom JDBC connector that can be used by JDBC-enabled applications to query and retrieve data from a custom data store. The custom JDBC connector is renamed and rebranded for your company and product.

Data Retrieval

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

- Each `IMetadataSource` implementation of `getMetadata()`
- `ULPersonTable.getData()`

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

In addition, both methods accept the following parameters:

- `data`

The `DataWrapper` into which you must copy your cell's value. This class is a wrapper around an Object managed by the SimbaEngine X SDK. 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.initializeColumns()` as `Types.INTEGER`, then when `ULPersonTable.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`).

- `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.

- `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.

Install the Evaluation License

You can use Simba SDK for 30 days after installing the evaluation license. The evaluation license is emailed to the person who registered the product.

Typically, you use Simba SDK to create your custom JDBC connector, then use a test JDBC-enabled application to retrieve data using the connector. Install the license file to the appropriate location for the type of JDBC-enabled application you are running.

Install the Evaluation License on Windows

To license Simba SDK for applications running under a non-SYSTEM user account:

- Save the license file in the `%USERPROFILE%` folder, for example:

  `C:\Users\<USER_ID>\SimbaEngineSDK.lic`

  Where *<USER_ID>* is the ID of the user running the application.

To license Simba SDK for applications running under a SYSTEM user account:

- For 32-bit applications on 32-bit machines, or 64-bit applications on 64-bit machines, save the license file as follows:

  `C:\Windows\System32\config\systemprofile\SimbaEngineSDK.lic`

- Or, for 32-bit applications running on 64-bit machines, save the license file as follows:

  `C:\Windows\SysWOW64\config\systemprofile\SimbaEngineSDK.lic`

Install the Evaluation License on Unix, Linux, and macOS

To license the Simba SDK:

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

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

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

Contact Us

For more information or help using this product, please contact our Technical Support staff. We welcome your questions, comments, and feature requests.

Note:

To help us assist you, prior to contacting Technical Support please prepare a detailed summary of the Simba SDK version and development platform that you are using.

You can contact Technical Support via the Magnitude Support Community at www.magnitude.com.

You can also follow us on Twitter @SimbaTech and @Mag_SW.

Third-Party Trademarks

Simba, the Simba logo, Simba SDK, and Simba Technologies are registered trademarks of Simba Technologies Inc. in Canada, United States and/or other countries. All other trademarks and/or servicemarks are the property of their respective owners.

Kerberos is a trademark of the Massachusetts Institute of Technology (MIT).

Linux is the registered trademark of Linus Torvalds in Canada, United States and/or other countries.

Mac and macOS are trademarks or registered trademarks of Apple, Inc. or its subsidiaries in Canada, United States and/or other countries.

Microsoft SQL Server, SQL Server, Microsoft, MSDN, Windows, Windows Azure, Windows Server, Windows Vista, and the Windows start button are trademarks or registered trademarks of Microsoft Corporation or its subsidiaries in Canada, United States and/or other countries.

Red Hat, Red Hat Enterprise Linux, and CentOS are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in Canada, United States and/or other countries.

Solaris is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

SUSE is a trademark or registered trademark of SUSE LLC or its subsidiaries in Canada, United States and/or other countries.

Ubuntu is a trademark or registered trademark of Canonical Ltd. or its subsidiaries in Canada, United States and/or other countries.

All other trademarks are trademarks of their respective owners.