Magnitude Simba SDK

Build a JDBC Driver 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 not SQL-capable. It explains how to customize the JavaQuickJson 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.

For data stores that do not support SQL, the Simba SDK provides an SQL parser and an execution engine. Developers can use these features to translate SQL queries to a custom API that the data store understands.

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 JavaQuickJson 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 not SQL-capable.

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`.

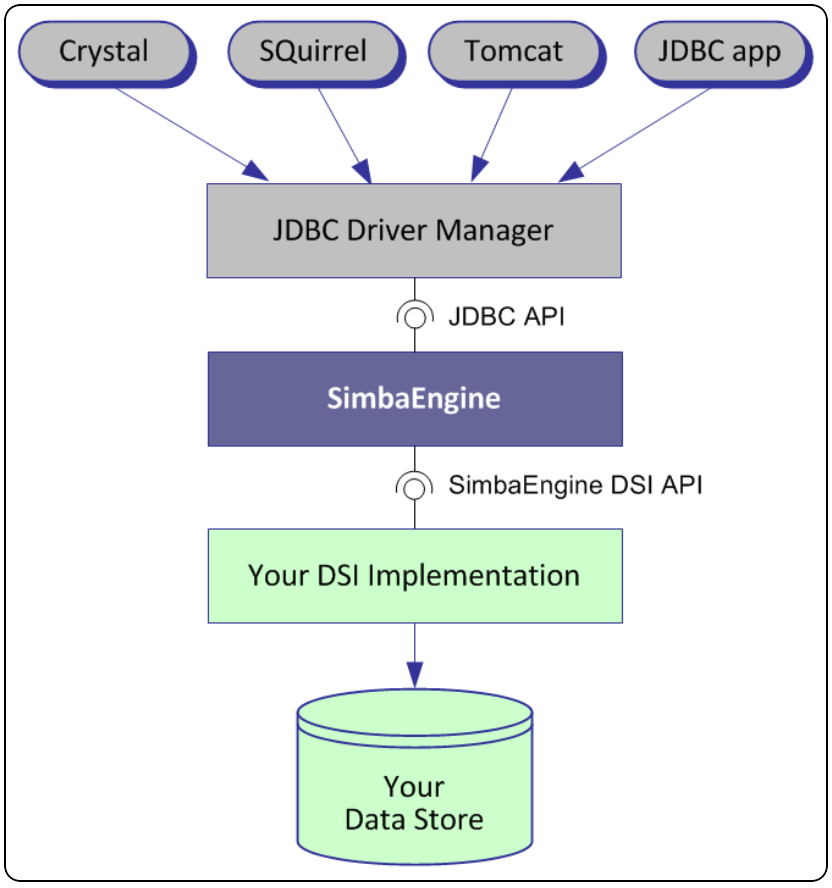
SimbaEngine X SDK Overview

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 JavaQuickJson Sample Driver

The sample projects included with the SimbaEngine X SDK are functioning DSI implementations that you can copy and modify to create your own driver for your own data store. This document explains how to use the JavaQuickJson sample driver to connect to an in-memory, hierarchical collection of data nodes, which are loaded from a JSON file.

Since JSON files are not an SQL-capable data store, the Simba Java SQLEngine component of the SimbaEngine X SDK must be used to perform the necessary SQL processing.

Note:

The Simba SQLEngine is a component of the SimbaEngine X SDK that you use to convert SQL commands into commands that your data store understands.

This guide explains how to make the following modifications to the JavaQuickJson 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 JavaQuickJson 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 JavaQuickJson driver, you can use it as the foundation for a commercial DSI implementation.

Sample JDBC Driver Architecture

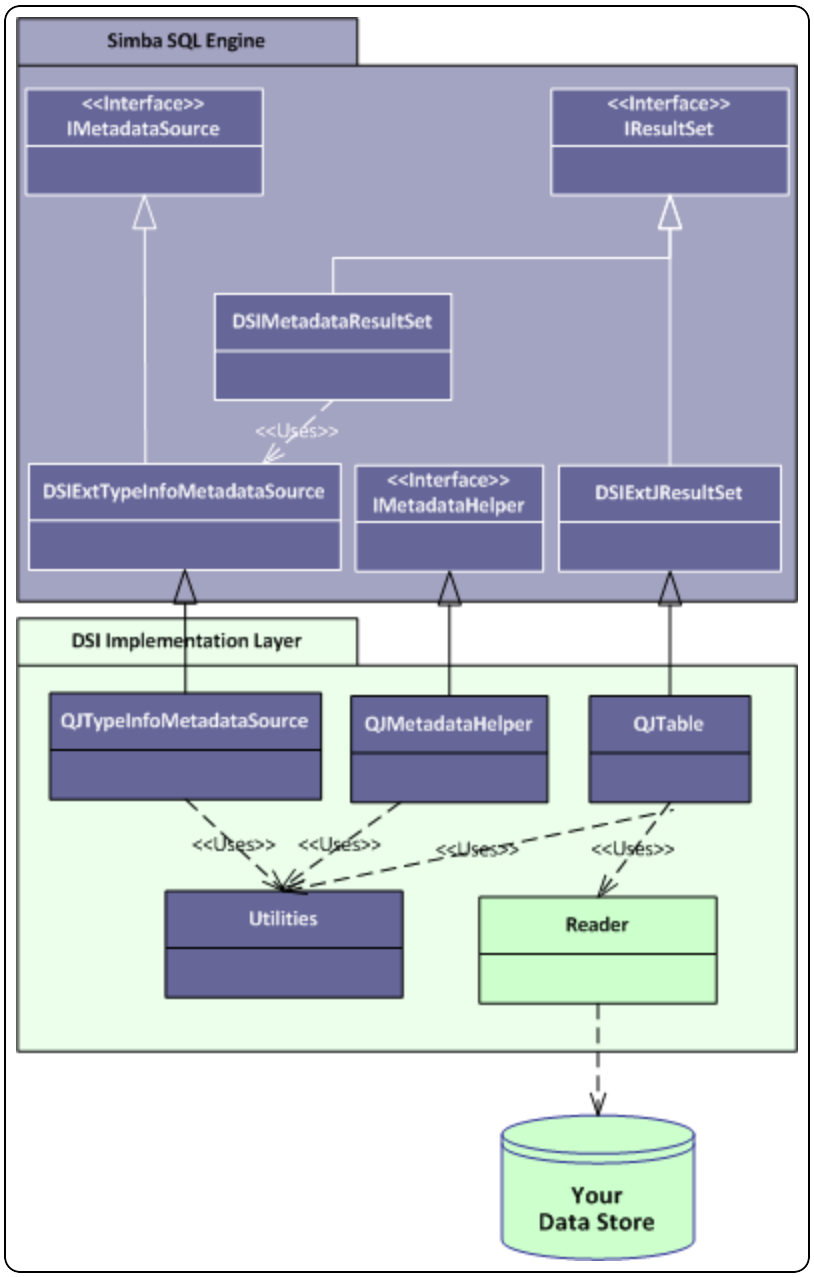
This section provides a technical overview of the sample JDBC driver that is provided with the SimbaEngine X SDK.

Note:

For detailed information on the SimbaEngine X SDK, see Developing Connectors for Data Stores Without SQL

The UML diagram below shows a common design pattern for a DSI built using the SimbaEngine X SDK. The diagram shows the main classes and interfaces for exposing and working with data and metadata in your data store.

The implementations for the JavaQuickJson sample are shown in the diagram below. Note that the concrete implementations are named with the prefix `QJ`. It is these concrete implementations that you will take and modify for your own data source.



`IResultSet` defines an interface for working with resultsets, including both data and metadata. It is responsible for retrieving column data and maintaining a cursor across result rows. Default implementations named with the prefix `DSI` are provided by the SDK and must be overridden and implemented in your driver. The SDK also provides a metadata helper interface which can be optionally implemented for quickly accessing metadata from a metadata resultset.

The `QJTable` class provides methods for accessing and manipulating a table of data in your data store. The `Reader` node used by `QJTable` represents all of the data store specific code used by `QJTable` to talk to the data store. For example, the data store in the JavaQuickJSON sample consists of JSON data files which are read and stored using a third party JSON parser called Jackson. Therefore the `Reader` node represents the Jackson parser while the icon entitled Your Data Store represents the underlying JSON files.

Implementing data retrieval is straightforward. Your `Reader` code interacts directly with your data store to retrieve the data in response to queries and to update the underlying data store on demand. The `Reader` should take care of caching, buffering, paging, and all the other techniques that speed data access.

Metadata access can be provided by either implementing your own `DSIMetadataSource`-derived classes or implementing an `IMetaDataHelper` for use by some of the metadata stores included with SimbaEngine X SDK to provide metadata information. As a starting point to make your driver work properly, your driver needs to provide metadata access for the following types:

- Tables
- Catalog only
- Catalog/Schema only
- Table Type Only
- Columns
- Type Information

JavaQuickJson also makes use of an internal helper class denoted by the `Utilities` node. In this sample driver, the `QJCoreUtils` class contains platform-specific utility functions used by the various classes in the driver.

If your resulting DSI implementation design looks like the image above after completing this guide, you are on the right track.

Note:

SimbaEngine versions 9.2 to 10.1, and 10.2.2 and above allows you to develop your driver to support multiple JDBC specifications by constructing a 'hybrid' driver that dynamically chooses the appropriate driver at runtime. This document will highlight the changes/differences related to this flexibility where applicable.

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, JavaQuickJson, 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 38

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 JavaQuickJson project as an existing project into your workspace.

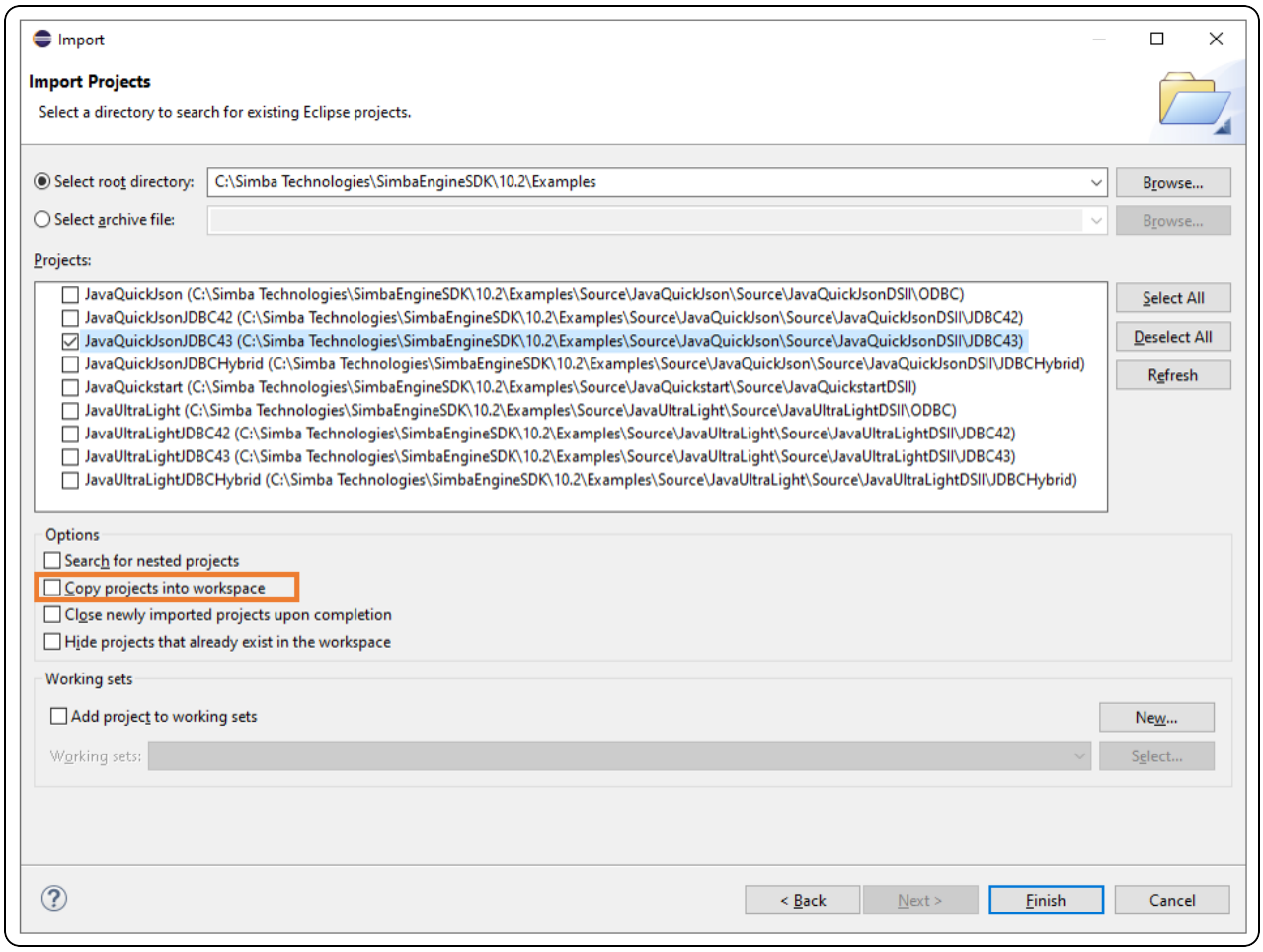
   The project files are located under `[INSTALL_DIR]\Examples\Source\JavaQuickJson\Source\JavaQuickJsonDSII\[JDBC_VERSION]`, where `[JDBC_VERSION]` is the version of JDBC that the sample driver supports. The hybrid project files are located under `[INSTALL_DIR]\Examples\Source\JavaQuickJson\Source\JavaQuickJsonDSII\JDBCHybrid`.

   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 **JavaQuickJsonJDBC43** project to build a sample Java driver that supports JDBC 4.3:



Note:

The SimbaEngine X SDK includes a sample driver called **JavaQuickStart**, which shows you how to write an ODBC driver using Java. This guide uses the **JavaQuickJson** driver, which shows you how to write a JDBC driver using Java.

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 **JavaQuickJson** project.
   b. Right-click on the **JavaQuickJsonBuilder*[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 **JavaQuickJsonBuilder*[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\JavaQuickJson\Lib`.

You have built the sample JDBC driver.

Test the Sample JDBC Driver

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

To test the sample driver:

1. Open DbVisualizer.
2. Add the JavaQuickJson driver that you built in Build the Sample JDBC Driver on page 14:
   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 14. For example: `[INSTALL_DIR]\Examples\Source\JavaQuickJson\Lib\JavaQuickJsonJDBC42.jar`. Then, click **Open**.
   g. Close the Driver Manager window.

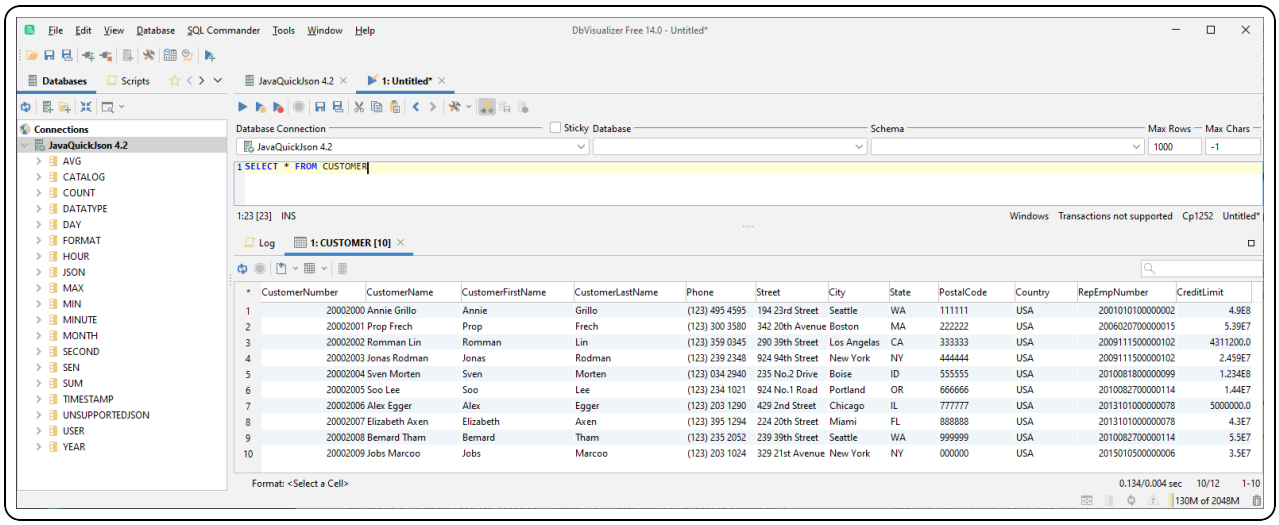
   The JavaQuickJson driver is added to the DbVisualizer application.
3. Add a connection to the driver:
   a. Select **Database** > **Create Database Connection** and choose the driver you added in the previous step.
   b. Enter the name of the connection (you can enter any name), then click **Next**.
   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;DBF=[INSTALL_
      DIR]\Examples\Databases\QuickJson
      ```

   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* FROM CUSTOMER.
   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* FROM CUSTOMER".
   d. Click the blue arrow to execute the statement.

The results, including CustomerNumber, CustomerName, and other columns, are displayed. For example:



Once the JavaQuickJson 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 `JavaQuickJson` folder and paste it to the same location. This will create a new directory called "JavaQuickJson - 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\JavaQuickJsonDSII\[JDBC_VERSION]\JavaQuickJsonBuilder[JDBC_VERSION].xml`

   b. Using a text editor, replace every instance of “JavaQuickJson” 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 “JavaQuickJsonJDBC*[JDBC_VERSION]*” to the name of your new JDBC driver.
   c. Update the path in the **linkedResources** to account for the directory name change performed in Step 1.
   d. Save and close the `.project` file.
4. Rename the `[INSTALL_DIR]\Examples\Source\<YourRenamedFolder>\Source\JavaQuickJsonDSII\CommonSource\JavaQuickJsonDriverBuilder.xml` file, replacing "JavaQuickJson” with the name of your driver chosen in the previous step.

   Important:

   We recommend that you name your project resulting binary (that is, the .jar file) to indicate the version of JDBC it supports. For example:

   - `MyProjectJDBC42.jar`
   - `MyProject_hybrid.jar`

5. Import and build the project to ensure it compiles without errors.
6. 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. (QJDriver.java)

TODO #2: Set the driver properties.	(QJDriver.java)
TODO #3: Set the connection properties.	(QJConnection.java)
TODO #4: Set the driver-wide logging details.	(QJDriver.java)
TODO #5: Set the connection-wide logging details.	(QJConnection.java)
TODO #6: Check Connection Settings.	(QJConnection.java)
TODO #7: Establish a connection to your data store.	(QJConnection.java)
TODO #8: Create and return your Metadata Sources.	(QJDataEngine.java)
TODO #9: Open A Table.	(QJDataEngine.java)
TODO #10: Update Messages properties file.	(QJDriver.java)
TODO #11: Register your error messages for handling by DSIMessageSource.	(QJDriver.java)
TODO #12: Set the vendor name for the error messages.	(QJDriver.java)
TODO #13: Update the component name.	(QuickJSON.java)
TODO #14: Assign a unique component ID value to the messages.	(QuickJSON.java)
TODO #15: Set the JDBC Component Name for error messages originating from the JDBC layer.	(QJJDBC42Driver.java, QJJDBC43Driver.java, or QJJDBCHybridDriver.java)
TODO #16 : Set the JDBC Component Name for error messages originating from the JDBC layer.	(QJJDBC42DataSource.java, QJJDBC43DataSource.java, or QJ JDBCHybridDataSource.java)
TODO #17: Set the subprotocol to which this driver will respond.	QJJDBC42Driver.java, QJJDBC43Driver.java, or QJJDBCHybridDriver.java)
TODO #18: Set the subprotocol to which this driver will respond.	(QJJDBC42DataSource.java, QJJDBC43DataSource.java, or QJ JDBCHybridDataSource.java)
TODO #19: Define your custom client info properties.	QJConnection.java
TODO #20: Implement the wanted behaviour.	QJConnection.java

7. Use any JDBC-enabled application to test your custom JDBC driver, as described in Test the Sample JDBC Driver on page 17.

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 `QJDriver` 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 17.

1. Set a breakpoint in your project.
   a. In Eclipse, open the file `com.simba.qjdriver.dataengine.QJDataEngine`.
   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 17.

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 JavaQuickJson sample connector.
  This verifies that your installation and development environment are properly configured.
- Created, built, and tested a custom connector project by copying the JavaQuickJson 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 “JavaQuickJson” 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 `QJDriver'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 `QJConnection'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 #19: Define your custom (QJJDBC42DataSource.java,

client info properties.	QJJDBC43DataSource.java, or QJJDBCHybridDataSource.java)
TODO #20: 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.	(QJDriver.java)
TODO #5: Set the connection-wide logging details	(QJConnection.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.	(QJConnection.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 `QJConnection.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 `QJConnection.connect()`, passing in all the connection settings received from the application.

TODO #7: Establish a Connection to your data store. (QJConnection.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.

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()

You can choose to implement an `IMetaDataHelper` class for some of the metadata sources, or you can extend `DSIMetadataSource`. You can also use a mixture of these approaches.

Using IMetaDataHelper or Extending DSIMetadataSource

Implementing an `IMetaDataHelper` class is convenient because it allows the SQLEngine to synthesize the metadata information for metadata stores that use it. However, using `IMetaDataHelper` can have slower performance than with a custom class derived from `DSIMetadataSource`. This is because `IMetaDataHelper` must iterate over each table to build the metadata while a custom `DSIMetadataSource` derived metadata source can be built around a specific table, and additional custom information can be returned only if needed.

In the JavaQuickJson driver, both solutions are used. An `IMetaDataHelper` implementation called `QJMetadataHelper` is passed to the SDK-provided metadata stores for most information (e.g. schemas) while a custom metadata source class called `QJTypeInfoMetaDataSource` provides type information.

TODO #8: Create and return your Metadata sources. (QJDataEngine.java)

`QJDataEngine.makeNewMetadataSource()` 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 37 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`, `QJDataEngine.makeNewMetadataSource()` will return an instance of `QJTypeInfoMetadataSource()`.
2. The SimbaEngine X SDK supports all required JDBC data types. The JavaQuickJson sample driver exposes support for the following types:
   - BIT
   - BIGINT
   - DOUBLE
   - WVCHAR
   - NUMERIC
3. For your driver, you may need to change the types returned and the parameters for the types in `QJTypeInfoMetadataSource's initializeDataTypes()`.

Handle Other Metadata Sources

The JDBC functions that handle metadata, including `getCatalogs()`, `getSchemas()`, `getTables()`, and `getColumns()`, are handled automatically by the metadata helper class.

When called with any other `MetadataSourceId`, `QJDataEngine's makeNewMetadataTable()` should return NULL. Returning NULL tells SimbaEngine X SDK that it should use the metadata helper class returned via `QJDataEngine's createMetadataHelper()`, along with some default `IMetadataSources`, to create the data store metadata.

The following changes are required:

1. `QJMetadataHelper.QJMetadataHelper()`

   The example constructor retrieves a list of the tables in the *data store*. You should modify this method to load the tables defined within your data store.
2. `QJMetadataHelper.getNextTable()`

   In the JavaQuickJson sample driver, this method returns the next table in the data store. You should modify this method to retrieve the next table from your data store.
3. `QJMetadataHelper.getNextProcedure()`

   This method returns the next procedure in the data store. If procedures are supported, you should modify this method to retrieve the next procedure from your data store.
4. Implement `QJTable`.

   The `DSIExtMetadataHelper` class works by retrieving the identifying information for each table and then opening the table via `QJDataEngine.openTable()`. Once you have implemented `QJTable`, the correct metadata will be returned for all of the tables and columns in your data store.

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.

Open a Table

The JavaQuickJson driver has implemented a simple in-memory table loaded from a JSON file. `QJDataEngine.openTable()` is the entry point where the Simba SQL Engine requests that tables involved in the query be opened.

TODO #9: Open a Table. (QJDataEngine.java)

QJDataEngine.openTable()

Modify `QJDataEngine.openTable()` to check that the supplied catalog, schema and table names are valid and correspond to a table defined in your data store. If they are not, you should return `null` to indicate that the table does not exist. If the inputs are valid, a new instance of `QJTable` is returned. In the JavaQuickJson driver, `openTable()` uses the `QJCoreUtils` class's `findTableInMapping()` method to get the specified table if it exists.

The QJTable class

`QJTable` is an implementation of `DSIExtJResultSet`, an abstract class provided by the SimbaEngine X SDK that provides the basic forward-only traversal of result sets The main role of `QJTable` is to translate the stored data from your native data format into SQL Data types.

See Modify the QJTable class for information on modifying this class to make it work with your data store.

Modify the QJTable class

The main role of `QJTable` is to translate the stored data from your native data format into SQL Data types. This section walks you through the required steps for modifying the class for your own driver.

Return the catalog, schema and table names for your table

Modify the following methods to return catalog, schema, and table names for your table

1. `QJTable.QJTable()`:

   The constructor must be modified to take in the catalog, schema and table names and save them in member variables.
2. `QJTable.getCatalogName()`:

   Returns the catalog corresponding to the table.
3. `QJTable.getSchemaName()`:

   Returns the schema corresponding to the table.
4. `QJTable.getTableName()`:

   Returns the name of the table.

Return the columns defined for your table

You must modify `QJTable.initializeColumns()` so that, for each column defined in the table, you define the `ColumnMetadata` in terms of SQL types. Note that in the JavaQuickJson driver, this method delegates some of this work to the `flattenArray()` and `flattenObject()` helper methods which, in turn, delegate the creation of ColumnMetadata to another helper method called `addColumnMetadata()`.

Example: Pseudocode for your new method

```
Get all the column information from your data store for the
table
For Each Defined Column
{
// Change the parameter of this method to the SQL Type that
// maps to your data store type.
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 );
}
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);
}

m_columns.add(columnMetadata);
}
```

Data retrieval

The following methods are used together for navigating a data structure containing information about one table in your data store, and retrieving data from that table.

- `QJTable.moveToNextRow()`:

This method is invoked when the next row of data is being requested. Most implementations will use this method to position an internal cursor on the next row of data.

- `QJTable.getData()`:

This method is invoked to return data from the current row and a column within that row. The method also takes in parameters for obtaining variable sized data.

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 JavaQuickJson Driver, `QJTable` stores its data in an in-memory graph of JSON node objects. Objects represent rows of data and child objects represent columns of data. The `QJTable.getData` method takes a column index and uses it to determine from which node of the current row/object to retrieve data. See Data Retrieval on page 37 for more information.

Close the connection

The `QJTable.closeCursor()` callback method is called from Simba SQL Engine to indicate that data retrieval has completed and that you may now do any tasks related to closing the connection to your data store. For example, the JavaQuickJson driver resets some internal members which reference the underlying JSON structure.

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.

Update the Error Messages

All the error messages used within your DSI implementation are stored in a file called `messages.properties`. For this optional TODO, you can modify the RESOURCE_NAME string to change the log file name.

TODO #10: Update Messages properties file.	(QJDriver.java)

Also, you should review each exception thrown within your DSI implementation and change the parameters to match.

Register the Error Messages

By default, the driver's `messages.properties` file resides in the same package as the `QJDriver` 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 #11: Register your error messages for handling by DSIMessageSource.	(QJDriver.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 #12: Set the vendor name for the error messages.	(QJDriver.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 "JavaQuickJsonDSII" to a

custom name. This rebranding identifies messages that are logged by the custom DSII.

TODO #13: Update the component name.	(QuickJSON.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 #14: Assign a unique component ID value to the messages.	(QuickJSON.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 #15 : Set the JDBC Component Name for error messages originating from the JDBC layer.	(QJJDBC42Driver.java, JDBC43Driver.java, or QJJDBCHybridDriver.java)
TODO #16: Set the JDBC Component Name for error messages originating from the JDBC layer.	(QJJDBC42DataSource.java QJJDBC43DataSource.java, or QJ 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 #17: Set the subprotocol to which this driver will respond.	(QJJDBC42Driver.java, QJJDBC43Driver.java, or QJJDBCHybridDriver.java)
TODO #18: Set the subprotocol to which this driver will respond.	(QJJDBC42DataSource.java, QJJDBC43DataSource.java, or QJ

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 “QuickJSON” to reflect your driver name. This is usually related to the name chosen in steps 2 and 3 on Day One.
- The letters “QJ” 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()`
- `QJTable.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 `QJTable`, 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 `QJTable.initializeColumns()` as `Types.INTEGER`, then when `QJTable.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.