Magnitude Simba REST SDK

Developing RESTful Connectors for REST based Datastores

Version 10.2

May 2023

Copyright

Contact Us

Magnitude Software, Inc.

www.magnitude.com

About This Guide

Purpose

This guide explains how to use the Magnitude Simba REST SDK to quickly and efficiently develop a RESTful ODBC Connector and connect to remote data stores via REST interface.

The Simba REST SDK includes REST features to let your connector communicate and get data from server via REST API. This guide walks you through the installation procedure for Simba REST SDK and the OneDrive Sample Connector. It also covers how to develop a Custom RESTful ODBC Connector using Simba REST SDK.

Audience

The guide is intended for developers who want to create a Custom RESTful ODBC Connector to connect to and get data from datasource that expose REST APIs with minimum time and effort. Simba REST SDK already provides major features to work with REST APIs, developers just need to configure the connector to use that feature provided by SDK. This guide is also intended for end users of the Simba SDK.

Knowledge Prerequisites

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

- Familiarity with the platform on which you are using the Simba REST SDK.
- Ability to use the data store to which the Simba REST SDK is connecting.
- An understanding of the REST APIs and how to use them to retrieve data from datastore.
- An understanding of the role of ODBC technologies and driver managers in connecting to a data store.
- Experience creating and configuring ODBC connections.

Variables Used in the Document

The following variables are used in this document:

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

Table of Contents

About This Guide 3

Variables Used in the Document 4

Specifications 8

Supported Platforms 8

Introducing Simba REST SDK 9

Creating a Custom Connector with the Simba REST SDK: 9

Core Features 12

Authentication 12

Pagination 12

Parsers 13

Multi-Threaded Data Retrieval 13

Data PreFetch support 13

Data Type Mapping 13

Tables & Columns 13

SkeletonTables & SkeletonColumns 14

LazyInitialization 14

Passdown 14

Throttling 15

Creating a Sample Driver 17

Install the Simba SDK for RESTful Datastores 17

Build the Sample RESTful ODBC Connector 18

Setup Registry for OneDrive Sample Connector 19

Connect to the Data Store 20

Set Up a Custom ODBC Connector Project 21

TODO #1:Set Driver wide Configuration 24

TODO #2: Set Connection wide Configuration 26

TODO #3: Set Sensitive Keys 27

TODO #4: Set Base Configuration 27

TODO #5: Setup the BrowseConnectMap to be used during Authorization process: 30

TODO #6: Configure Authentication & Parser: 31

TODO #7: Configure Tables & Columns: 39

Passdown 60
PreReqCalls 62
Skeleton Tables & Skeleton Columns 67
Custom Extensions 72
- Authentication 72
- Pagination 80
- Parser 95
- Modify HTTP Request / Response 100

Connector Configuration Options 103
- Access Token 103
- Auth_Type 103
- Client ID 104
- Client Secret 104
- ConnIdentifier 104
- DisableColumnPushdown 105
- Driver 105
- EnableURLLog 105
- EncClientSecret 106
- EncodingType 106
- EncryptSwapFile 106
- HideSchemas 107
- Host 107
- LazyInitialization 108
- Log Level 108
- Log Path 109
- Max File Size 109
- Max Number Files 110
- MaxOffset 110
- MemoryManagerMemoryLimit 110
- MemoryManagerStrategy 111
- MemoryManagerSwapDiskLimit 112
- MemoryManagerThresholdPercent 112
- Minimum TLS 112
- MinOffset 113

Password 113
Proxy Host 113
Proxy Password 114
Proxy Port 114
Proxy Uid 114
RequestTimeout 114
Swap File Path 115
Use HTTPS 115
UseHostVerification 116
UsePeerVerification 116
User 116
UseThrottling 117
UseWindowsProxySettings 117

Contact Us 118

Third-Party Trademarks 119

Specifications

This section lists the platform and compiler requirements for the Simba REST SDK.

Supported Platforms

This section lists the platforms and compilers that are supported by the Simba REST SDK version 10.2.

Hardware Requirements

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

- 8 GB of free disk space
- 1 GB RAM

Software Requirements

The following table lists the supported platforms and compilers:

Platform	Versions	Compilers	Bits
Windows	10 & 11 Server 2016, 2019 & 2022	Visual Studio 2015, 2017, 2019, & 2022	32, 64

Introducing Simba REST SDK

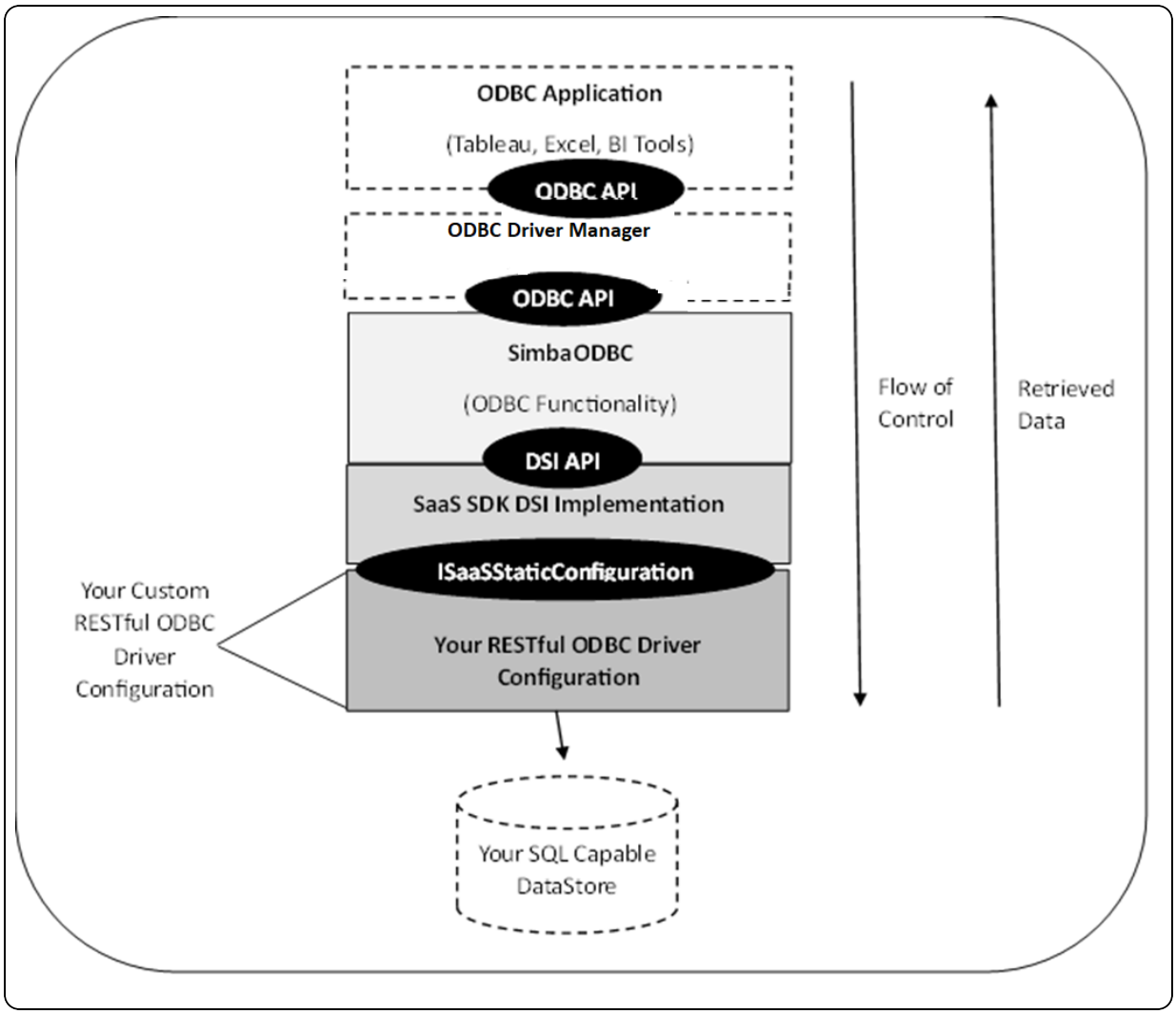
Simba REST SDK is a collection of tools used to create RESTful ODBC Connectors. Custom RESTful ODBC Connectors can be built for any aata source that exposes REST APIs to get data in the form of JSON, XML or CSV.

This guide explains how to use SDK features to create a Custom RESTful ODBC Connector used for data analysis and other applications where ODBC Connector is used as a primary source of data.

Creating a Custom Connector with the Simba REST SDK:

The components of Simba REST SDK implement all the necessary functionality to work with the REST API: Authentication, Pagination, Form HTTP Request , Parse HTTP Responses (in form of JSON, XML, CSV), Error and Exception handling, maps the data got from REST API to SQL output as per Driver Configuration.

Below is the high-level architecture of where Simba REST SDK Fits into the Simba SDK architecture.



ISaaSStaticConfiguration

`ISaaSStaticConfiguration` defines a generic view of Driver Configuration. It includes classes and methods to configure data source metadata, Driver Wide Configuration (available throughout Driver scope), Connection Wide Configuration (available throughout an ODBC Connection).

Simba REST SDK DSI Implementation:

Simba REST SDK abstracts the standard DSI implementation from Driver Configuration to map DSI with your datastore. To know more about DSI, please refer sub section, *Data Store Interface Implementation* (DSII) under head section *Creating a Custom Connector with the Simba SDK* in Simba SDK Developer Guide SQL Enabled guide.

Note:

Simba REST SDK DSI Implementation already implements the DSI implementation layer for your connector. This means you can just focus on configuring the data source specific information in Connector Configuration.

Getting Started with the Sample Connector Projects

The sample connector projects are a great way to get started developing your custom connector. This guide walks you through the steps of building, configuring, and customizing the project.

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

OneDrive Sample Connector

OneDrive is a C++ sample RESTful ODBC connector that connects to and get Data from OneDrive via REST API and Output the data in SQL-92 format. This sample's purpose is to provide a simple, working connector that you can refer and transform into a connector that is configured to work with your desired Datastore.

Core Features

Authentication

ODBC Applications first need to authenticate with the REST based datastore before fetching any data. The most common authentication mechanisms supported by modern REST based datastores are OAuth 2.0, Basic Authentication, Access Token based authentication. These authentication mechanisms are already supported and implemented by SDK. You just need to provide a few configuration details required for authentication to work. See Configure Authentication & Parser for more details under creating a sample driver. In addition to this, SDK provides an interface extension, refer Authentication under Custom Extensions on page 72 which can be used to implement custom authentication for specific authentication type not supported by SDK by default.

Pagination

REST APIs support paginating the data across multiple API requests to control the amount of data fetched from each API request. This helps to divide the data traffic across multiple HTTP requests sent to server which otherwise will cause more network delay and more traffic using the single HTTP request.

SDK implements the most common pagination types including:

Index Based Pagination

Pages are identified by using a unique identifier. For example, Offset and each page is assigned a pagesize and number of objects to be returned to limit the amount of data to be returned. The total count of objects is provided by the REST API in the API response which can be referred to determine when to terminate the pagination. In case total count is not exposed in the API response, the pagination termination is determined by the empty page. For example empty response which means that no more objects left to be returned by API.

Response Based Pagination

Each page is assigned a pagesize. For example, number of objects to be returned to limit the amount of data to be returned. API provides the link to the next page in the response body of the current page or response headers of the current page or it provides a NextPageToken in the response body of the current page which when supplied to the next API request, the next page will be returned from server. If NextPageURL or NextPageToken is not present in the response body or response headers of the current page, it indicates end of pagination.

In addition to this, SDK provides the interface to pagination to implement Custom Extensions on page 72 for the pagination type not supported by SDK by default.

Parsers

Parsers the API response data in desired formats. For example, XML, JSON, CSV and so on. For each response data format, a separate parser is supported by SDK. For example, for JSON response data, JSON parser is supported. SDK supports parsers for 3 response data formats: JSON, XML, CSV. Parser parses the response data and gets the rows of data out of it as per the columns metadata configuration.

SDK Provides the interface to Parser Extension to implement Custom Extensions on page 72 parser for response types not supported by SDK by default.

Multi-Threaded Data Retrieval

If possible, data can be retrieved from server by sending multiple REST API requests at the same time which improves performance while retrieving huge number of records from the server over multiple API requests. This is possible if table is configured to use INDEX_BASED_PAGINATION where indexes can be precalculated based on the total count, for example, total no. of records and value received from the response of the first page. How many requests can be sent parallely is controlled by the connection parameter MaxThreads.

Data PreFetch support

A REST API endpoint may require some specific data to execute which can be pre fetched by using another REST API endpoint (also called `PreReqCall`. Refer section PreReqCalls on page 62 for more details) executed prior to the execution of the final endpoint to get the required data to be added into the final REST endpoint. Then the execution of final REST endpoint gives us the intended data output.

Data Type Mapping

Data source supported data types are not always SQL-92 types. We can map the data source supported type to that of SQL-92 by using this feature. This is useful in case we configure `SkeletonColumns` and the datatype of a column is identified from data source end which is not SQL-92 supported.

Tables & Columns

Tables are represented as a REST API object and columns as properties of that REST object. Consider an example of Get Orders API, it fetches information about orders with properties, for example, OrderId, OrderDate, ProductDetail, BillDetail. Here the OrderId and OrderDate are considered as columns for Orders table.

Depending on the requirement,

- the ProductDetail and BillDetail can be made separate tables, for example,Virtual Tables, with their own set of columns or
- the ProductDetail and BillDetail can act as columns for the same Orders table.

SkeletonTables & SkeletonColumns

Some data sources also expose REST API to get the metadata of REST objects and REST object properties,often called Custom Objects and its properties, which can be leveraged to define the metadata for tables and columns in your custom connector. SDK supports configuring the metadata for tables and columns using REST API and the tables and columns configured are called `SkeletonTables` & `SkeletonColumns`.

LazyInitialization

It specifies how we want SDK to initialize the tables, columns and other metadata. By default, SDK Initializes all the tables and columns metadata at the time of connecting to your data source using ODBC connector. It involves sending the REST API requests:

- to test connect with the data source.
- to Initialize the SkeletonTables and Columns (if any).
- to initialize other metadata (if any).

Sometimes this operation takes lot of time to initialize all metadata and requires the user to wait for that much interval of time which is not an Ideal Solution as it may initialize the information that might not be useful for the user for a particular Connection session.

Setting LazyInitialization flag enables the SDK to initialize the metadata for tables, columns, and other metadata only when required! for example, only if that table or column is used to get data or metadata or, SQLPrimaryKeys, SQLForiegnKeys, SQLColumns. And hence it reduces the TAT (Turnaround Time) by omitting the extra initialization that is not required to be initialized upfront. See Base Configuration section under Creating a Sample Driver on page 17 to see how to set this setting in your connector.

Passdown

REST APIs support filter or sort data based on the parameters passed to API request. To filter or sort data, we can passdown the column used to filter or sort data to an endpoint to let the server handle filter or sort operation and give us the filtered or

sorted result in the API response. Passdown feature lets you specify using which columns of a particular table, filter or sort operation is supported by REST API endpoint used for that table.

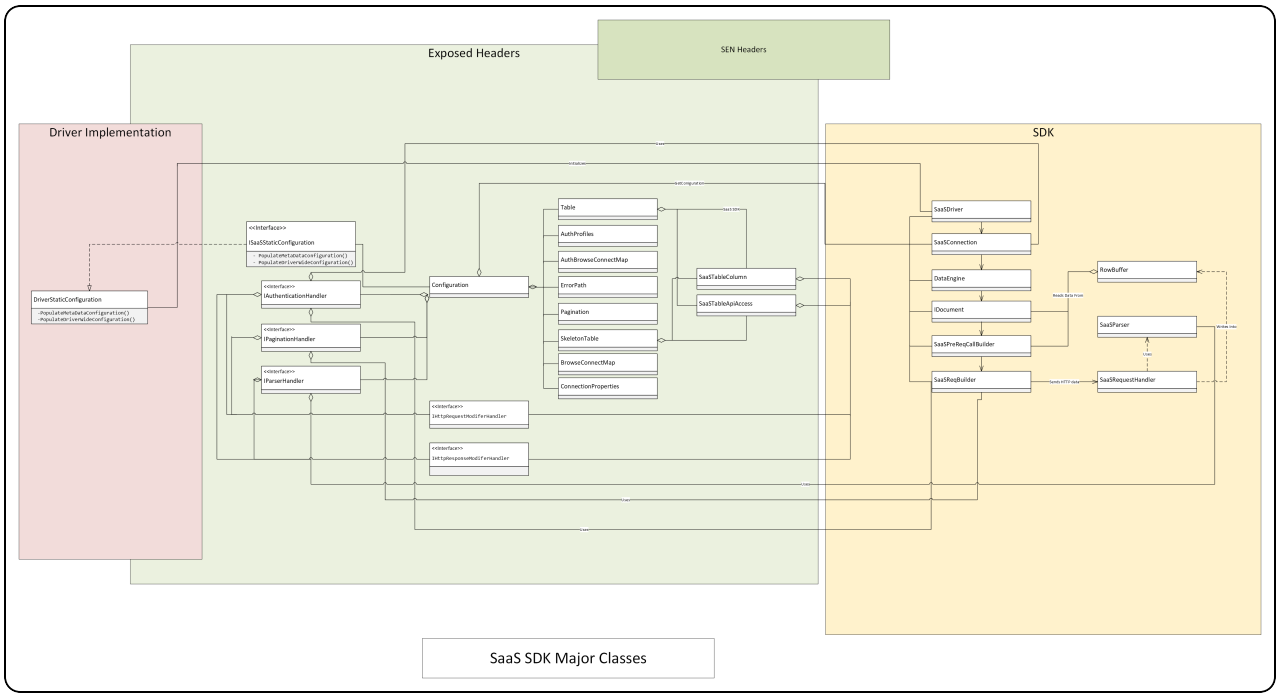
Maximum usage of passdown feature is recommended wherever possible to improve your connectors performance since we get the filtered result from the server response directly which otherwise would require first to fetch all data from server and let the Simba Engine SDK filter that data and give us the filtered output. See Passdown on page 60 section for more details).

Throttling

Data sources may impose limits on the no. of REST API requests allowed to send in one day or limits on the speed of requests at a time. If an API Client crosses these limits, it results in the rate limit errors which will cause the client to wait for some time interval and send the API requests again.

The speed at which each request is sent can be controlled by keeping a gap of some milliseconds between sending two API requests to server such that it never results in the rate limit error from server. The throttling support can be enabled in a connector by setting the “UseThrottling” connection parameter to value one in connection string and setting the throttling support to true in connector configuration level. See section TODO #4: Set Base Configuration under Creating a Sample Driver on page 17.

Below is the class diagram of the important headers of Simba REST SDK. Exposed headers are the one exposed to the end user for their custom implementation of driver and SDK headers are for internal use for SDK.



SaaS SDK Major Classes

Creating a Sample Driver

This Sample Driver guide walks you through the major Building Blocks of writing a Connector from Scratch using REST SDK and it also highlights the important features of SDK that you can use.

Instructions in this guide explain how to install the Simba SDK for RESTful datastores, compile the sample RESTful ODBC connector, and review the configuration information created at compile time.

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

After going through this section, you will have compiled, built and tested your custom RESTful ODBC connector.

Install the Simba SDK for RESTful Datastores

The Simba SDK for RESTful Datastores installation package includes:

- Simba SDK for RESTful Datastores implementation in Windows platform.
- Sample connector for Windows platform.
- PDF versions of the documentation.
- An HTML version of the C++ API.

To Install the Simba SDK for RESTful Datastores:

- Uninstall any previous versions of the Simba SDK for RESTful Datastores.
- Make sure that Simba SDK is installed in your system. Refer sub section *"Install the Simba SDK"* under head section "*Day One*" in guide "Build a C++ ODBC Driver in 5 Days (Windows).pdf" to install Simba SDK in your system.
- Double-click the Simba SDK for RESTful datastores executable that corresponds to your version of Visual Studio, and then follow the instructions provided in the installation wizard.

The installer sets the following environment variables, where *[INSTALL_DIR]* is the Simba SDK installation directory:

Environment Variable	Value
SIMBAREST_DIR	`[INSTALL_DIR]\SIMBARESTSDK\10.2\DataAccessComponents`
SIMBAREST_THIRDPARTY_DIR	`[INSTALL_DIR]\SIMBARESTSDK\10.2\DataAccessComponents\Thirdparty`

Important:

The Simba REST SDK environment variables are defined only for the user who ran the installation. If the SDK is installed as a regular user, Visual Studio must also be run as a regular user and not an administrator.

Build the Sample RESTful ODBC Connector

The OneDrive sample connector is included with the installation of the Simba REST SDK. It demonstrates one possible implementation of a read-only connector that reads data from OneDrive through REST API.

To build the OneDrive sample connector:

1. In Microsoft Visual Studio 2015, click **File** > **Open** > **Project** or **Solution**.
2. In the **Open Project** dialog, navigate to the following folder:
   a. *`[INSTALL_DIR]\SIMBARESTSDK\10.2\Examples\Source\OneDrive\Source`*
   Where `[INSTALL_DIR]` is the installation directory.
3. Select the file `SimbaRESTSDK_OneDrive_vs2015.sln`, and then click **Open**.
4. Click **Build** > **Configuration Manager**.
5. Click the drop-down arrow next to the **Active Solution Configuration** field, then select **Debug**, and click **Close**.
6. Click the drop-down arrow next to the **Active Solution Platform** field.
7. To build a 32-bit connector, select **Win32**.
8. To build a 64-bit connector, select **x64**.
9. Click **Close**.
10. Click **Build** > **Build Solution**.

The build appears in the following folder:

```
[INSTALL_
DIR]\SIMBARESTSDK\10.2\Examples\Source\OneDrive\Bin\<BUILD>\<R
ELEASE|DEBUG><CONFIGURATION>
```

Where:

- <BUILD> is a combination of your operating system, machine bitness, and compiler.methods
- <RELEASE|DEBUG> is release or debug
- <CONFIGURATION> is mt

For example:

```
C:\SimbaTechnologies\SIMBARESTSDK\10.2\Examples\Source\OneDriv
e\Bin\Windows_vs2015\debug32mt\SIMBARESTSDK_OneDrive_
Driver.dll
```

Setup Registry for OneDrive Sample Connector

Double click on one of the "Setup-<bitness>bitDriverOn<MachineBitness>bitWindows.reg" located in "[INSTALL_DIR]\SIMBARESTSDK\10.2\Examples\Source\OneDrive\Source" folder as per your requirement Where:

Bitness is the driver bitness you want to install (i.e. 32 bit Driver or 64 bit Driver).

MachineBitness is the bitness of the machine you'll install Driver in.

It will setup Registry for OneDrive ODBC connector in Windows Registry at:

- HKEY_LOCAL_ MACHINE\SOFTWARE\ODBC\ODBCINST.INI\Simba OneDrive ODBC Driver for 64- bit connectors on 64-bit machines, and 32-bit connectors on 32-bit machines.
- HKEY_LOCAL_ MACHINE\SOFTWARE\WOW6432NODE\ODBC\ODBCINST.INI\ Simba OneDrive ODBC Driver for 32-bit connectors on 64-bit machines.

To view the registry keys for the OneDrive connector:

- From a command line, run regedit.exe.
- In the registry editor, navigate to one of the following root directories:HKEY_LOCAL_MACHINE\SOFTWARE\ODBC for 64-bit connectors on 64-bit machines and 32-bit connectors on 32-bit machines.Or, HKEY_LOCAL_

`MACHINE\SOFTWARE\WOW6432NODE\ODBC` for 32-bit connectors on 64-bit machines.

- View the registry keys as explained in the rest of this section.

ODBC\ODBC.INI\Simba OneDrive key

This key defines the Data Source Name (DSN) for the OneDrive connector. It is located in the Windows Registry at:

- `HKEY_LOCAL_MACHINE\SOFTWARE\ODBC\ODBC.INI\Simba OneDrive` for 64-bit connectors on 64-bit machines, and 32-bit connectors on 32-bit machines.
- `HKEY_LOCAL_ MACHINE\SOFTWARE\WOW6432NODE\ODBC\ODBC.INI\Simba OneDrive` for 32-bit connectors on 64-bit machines.

This key has all the values that are needed to configure authentication, connection and driver behavior.Customize those values to provide your own OneDrive application details (i.e., Client Id, Client Secret, Tenant ID etc). Once you register your application for OneDrive with Microsoft, you get these details to be used for authentication purposes and work with REST API. If you haven't registered an application for OneDrive API, refer this Microsoft Docs link https://learn.microsoft.com/en-us/onedrive/developer/rest-api/getting-started/app-registration?view=odsp-graph-online.

To see the list of all properties common across all the connectors, check Connector Configuration Options on page 103 and configure it at this registry location.

Note:

Only the Driver subkey is required for your custom connector. Other sub keys are specific to the OneDrive connector.

Connect to the Data Store

To connect to the data store and test the OneDrive connector, any ODBC application can be used. This section shows how to use the ODBCTest tool, which is included in the Microsoft Data Access (MDAC) 2.8 Software Development Kit (SDK): http://www.microsoft.com/downloads/details.aspx?FamilyID=5067faf8-0db4-429a-b502-de4329c8c850&displaylang=en

To connect to the data store using the OneDrive connector:

1. Navigate to the folder containing the ODBC Test application, by default:`C:\Program Files (x86)\Microsoft Data Access SDK`

`2.8\Tools`

2. Navigate to the folder that corresponds to your connector's architecture: amd64, ia64 or x86.Example:If you built the 32-bit version of your connector on a 64-bit machine, select the x86 version.
    - Example:If you built the 32-bit version of your connector on a 64-bit machine, select the x86 version.
3. Click on odbcte32.exe to launch the ANSI version. Or, odbct32w.exe to launch the Unicode version.

> **Important:**
>
> It is important to run the correct version of the ODBC Test tool for ANSI or Unicode and 32-bit or 64-bit.

4. In the ODBC Test tool, click **Conn** > **Full Connect**. The Full Connect window opens.
5. In the Full Connect dialog, select "Simba OneDrive" from the list of data sources, and then click **OK**.
6. In the ODBC Test window, enter `SELECT * from DriveRoot`.
7. Click  and  to output a simple result set. The results are displayed in the window.

   You have successfully used the OneDrive connector to connect to the sample data store and retrieve data.

Set Up a Custom ODBC Connector Project

Once the OneDrive has been built and tested, you can create a new project for your custom RESTful ODBC connector.

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 a custom project:

1. In Windows Explorer, copy the following directory and paste it to the same location:
   a. *[INSTALL_DIR]\Examples\Source\OneDrive.*
   b. where *[INSTALL_DIR]* is the Simba REST SDK installation directory. This will create a new directory called `OneDrive - Copy`.
2. Rename the directory to the name of your custom ODBC connector. This name will be referred to as [*PROJECT*] for the rest of these steps.
3. Rename the file [*PROJECT*] `> Source > SimbaRESTSDK_OneDrive_vs2015.vcxproj`. This is the project file for your custom ODBC connector.
4. Rename the `.sln` file. This is the solution file for your custom ODBC connector.
5. Using a text editor, open the project file `.vcxproj` and replace every instance of OneDrive in the source code with the name of your custom ODBC connector.
6. Save and close the file `.vcxproj`.
7. Using a text editor, open the solution file `.sln` and replace every instance of OneDrive in the source code with the name of your custom ODBC connector.
8. Change any references to the project file to [*PROJECT*]`.vcxproj`.
9. Save and close the file.
10. Rename all files in Source or Resources folder and also open one by one in a text editor and replace OneDrive with the name of your custom ODBC connector.
11. Open `.sln` file in Visual Studio and replace OneDrive with the name of your custom ODBC connector in whole Project.

Build the Custom ODBC Connector

To build the custom ODBC connector:

In Microsoft Visual Studio 2015:

1. Click **File** > **Open** > **Project** or **Solution**.
2. In the Open Project dialog, navigate to the following folder: *[INSTALL_DIR]*\SIMBARESTSDK\10.2\Examples\Source*<PROJECTNAME>*\Source Where [INSTALL_DIR] is the installation directory.
3. Select the file <PROJECT>_vs2015.sln, and then click **Open**.
4. Click **Build** > **Configuration Manager**.
5. Click the drop-down arrow next to the **Active Solution Configuration** field, then select **Debug**, and click **Close**.
6. Click the drop-down arrow next to the **Active Solution Platform** field.
7. To build a 32-bit connector, select **Win32** and to build a 64-bit connector, select **x64**.
8. Click **Close**.
9. Click **Build** > **Build Solution**.

The build appears in the following folder:

[INSTALL_DIR]\SIMBARESTSDK\10.2\Examples\Source\<PROJECTNAME>\Bin\<BUILD>\<RELEASE|DEBUG><CONFIGURATION>,

Where:

<BUILD> is a combination of your operating system, machine bitness, and compiler

<RELEASE|DEBUG> is release or debug

<CONFIGURATION> is mt

For example:
C:\SimbaTechnologies\SIMBARESTSDK\10.2\Examples\Source\<PROJECTNAME>\Bin\Windows_vs2015\debug32mt\SIMBARESTSDK_<PROJECT>_Driver.dll.

Where:

[PROJECTNAME] is the name of the project folder you kept under Examples directory.

[PROJECT] is the name of the project you set in vcxproj and .sln files.

Once the Custom ODBC Connector is built successfully, go through the TODO messages to continue the development.

TODO #1:Set Driver wide Configuration

`SaaSDriverConfig` class provides all the global fixed configurations related to configuration valid at driver level, like drivername, datasource name, version number etc. This configuration is reused for all the connections created. Under the definition of method:

```
virtual void PopulateDriverWideConfiguration(SaaSDriverConfig&
out_driverConfig);
```

of your Driver's implementation of `ISaaSStaticConfiguration`, use the instance of `SaaSDriverConfig` class (`out_driverConfig`) to set the below driver wide information.

> **Note:**
>
> These details will be used to identify your Connector while Logging, INI File configuration, ETW Logging. So please provide all these details. For more details refer sub sections *"Rebranding your Connector", "Using INI files for Connector Configuration on Windows" and Logging to Event Tracing for Windows (ETW)" under head section "Productizing your Connector"* in Simba SDK Developer Guide SQL Enabled guide.

Driver name

The driver name to be used to represent the driver, this value is returned during the `SQLGetInfo`. Use the following method to set the driver name:

```
void SaaSDriverConfig::SetDriverName(const simba_wstring& in_
driverName)
```

Vendor name

Vendor name to be used for the driver, this can be company name using the driver. Use the following method to set the vendor name:

```
void SaaSDriverConfig::SetVendorName(const simba_wstring& in_
vendorName)
```

Datasource name

Actual data source name to identify the driver. For instance, if you are building driver for Salesforce Datasource, then datasource name can be set to Salesforce while driver name can be set to any customized name like Salesforce ODBC Driver. Use the following method to set the datasource name.

```
void SaaSDriverConfig::SetDataSourceName(const simba_wstring&
in_dataSourceName)
```

Driver version

The driver version is to be presented in four parts (major, minor, patch, build). Use the `ProductVersion` structure to mention version number and set it to driver wide configuration using the following method:

```
void SaaSDriverConfig::SetVersion(const ProductVersion& in_
dataSourceName)
```

If you don't set the driver version, it defaults to 1.0.0.1000.

ErrorMessage filename

SDK uses the error messages file (XML format) to read generic error messages that will be thrown for generic error cases , for example, not for Datasource specific errors, check `SaaSErrorPaths` to mention Datasource specific error message paths. Use the following method to set the name of error message file name:

```
void SaaSDriverConfig::SetErrorMsgFile(const simba_wstring&
in_errMsgFile)
```

Component Identifier

Component identifier for SIMBARESTSDK errors. If no component identifier is supplied, it defaults to 0 and the error message file is ignored or not used. Use the following method to set the component identifier for errors:

```
void SaaSDriverConfig::SetComponentIdentifier(simba_int32 in_
componentIdentifier)
```

Component name

Component name used in the log file for the driver implementation error messages. Use the following method to set the Component name:

```
void SaaSDriverConfig::SetComponentName(const simba_wstring&
in_componentName)
```

Driver GUID

Driver's unique GUID. This is used as a provider GUID in ETW Logging. Use the following method to set the Driver GUID:

```
void SaaSDriverConfig::SetDriverGUID(const GUID& in_
driverGUID)
```

An example implementation of `PopulateDriverWideConfiguration` method can be found in `DriverWideConfiguration.cpp` source file of sample driver.

TODO #2: Set Connection wide Configuration

Set the connection wide configuration like Catalog support, schema support. Catalog name, DBMS name. To know what all properties can be set at connection level, look for

`Simba::SaaSSDK::SaaSConnPropertyKey and Simba::SaaSSDK::SaaSConfigDataTypes` in SAAS SDK C++ API reference.

Implement the following method of `ISaaSStaticConfiguration` interface in your derived class to set the Connection wide Configuration for individual or all connections.

```
virtual void PopulateConnectionConfiguration(
                    const simba_wstring& in_connIdent,
                    SaaSConnectionConfig& out_connConfig);
```

You can set the connection properties for Individual connection by using the in_ connIdent parameter to identify each connection uniquely which is passed as a connection parameter in the Connection String used to uniquely identify each ODBC Connection. For more details check ConnIdentifier on page 104.

To set the new property, use `Simba::SaaSSDK::SaaSConnectionConfigValue` and pass the following values in its constructor:

```
SaaSConnectionConfigValue(
                SaaSConnPropertyKey in_key,
                SaaSConfigDataTypes in_dataType,
                void* m_data);
```

Once the property is set, it can be added into SaaSConnectionConfig by using the following method:

void AddToConnConfigMap(SaaSConnectionConfigValue* io_mapValue)

For more details, check *`Simba::SaaSSDK::SaaSConnectionConfig, and Simba::SaaSSDK::SaaSConnectionConfigValue`* in *REST SDK C++ API reference*.

For example, check the implementation of `PopulateConnectionConfiguration` method in `DriverWideConfiguration.cpp` source file in Sample Driver project.

TODO #3: Set Sensitive Keys

A Connection parameter key used in Connection string to specify any credential value could be a sensitive key such that it's value should be Masked in the Log files. To specify which all values in Connection String are Sensitive, implement the following method `of ISaaSStaticConfiguration` interface in your derived class:

```
virtual void PopulateSensitiveList(std::vector<simba_wstring>&
out_sensitiveList);
```

Push back the Connection parameter Keys in `out_sensitiveList` vector for the values which are sensitive in your connection string and should be masked in the log files.

An example implementation can be found in `PopulateSensitiveList` method implementation under `Configuration.cpp` source file of Sample Driver.

TODO #4: Set Base Configuration

Base Configuration details are details configured with REST API, for example, Base URL, Test URL Endpoint and used at Driver wide level. Implement the method `PopulateMetadataConfiguration` of `ISaaSStaticConfiguration` interface in your derived class to set the Base Configuration details for your Custom Connector.
`virtual void PopulateMetaDataConfiguration(SaaSConfiguration&
out_configs);`

Use the methods mentioned below of SaaSConfiguration to set the Base details:

- `void SaaSConfiguration::SetDataSource(const Simba::Support::simba_string &value)`

  Use this method to Set the Data Source name. For eg. OneDrive sample Connector sets the Datasource name to OneDrive using:

  ```
  out_configs.SetDataSource("OneDrive");
  ```

  Where out_configs is the instance of SaaSConfiguration where the Base level configs need to be set.

- `void SaaSConfiguration::SetBaseUrl(const Simba::Support::simba_string &value)`

  Use this method to set the base URL for your custom connector. The part of URL which remains unchanged across all REST API Endpoints supported by your custom connector is a base URL. Forexample, OneDrive sample connector sets the base URL as:

```
out_configs.SetBaseUrl("https://<HOST_PORT>");
```

This means SDK picks up the value of HOST and PORT Connection parameters at runtime as a BaseURL.

- `void SaaSConfiguration::SetTestUrlEndpoint(const Simba::Support::simba_string &value)`

  Use this method to set the Test URL Endpoint which will be used to do Connection Test for your Custom ODBC Connector. For eg. OneDrive Sample Connector sets the TestURLEndpoint as:

  ```
  out_configs.SetTestUrlEndpoint
  ("/api/REST/2.0/data/visitors");
  ```

  So the Connection Test URL will be formed as “https://<HOST_PORT>/api/REST/2.0/data/visitors” which if hit successfully by getting 200 (OK) response from server means that Connection test succeeded.

- `void SaaSConfiguration::SetTimestampFormat(const Simba::Support::simba_string &value)`

  Use this method to set the Time Stamp format in which the connector will return the TimeStamp values. For eg. OneDrive Sample connector sets the TimeStampFormat as:

  ```
  out_configs.SetTimestampFormat("y-MM-dd'T'HH:mm:ss.SSS");
  ```

  Where,

  Y stands for Year

  M stands for Month

  D stands for Date

  HH stands for hours in 24-hour format

  mm stands for minutes.

  ss stands for seconds.

- `void SaaSConfiguration::SetIsUnixTimeStampFormat(const bool &value)`

Use this method to tell SDK whether the TimeStamp received from DataSource Server is in UnixTimeFormat. Possible values are true, false. Default value is `false`.

If timestamp is in Unixtimeformat then SDK automatically converts it into the TimeStampFormat set using method SetTimeStampFormat. For example, OneDrive sample connector sets IsUnixTimeStampFormat to true using:

```
out_configs.SetIsUnixTimeStampFormat(true);
```

- `void SaaSConfiguration::SetTimestampUnit(const TimeStampUnit &value)`

  This defines the unit of UnixTimeStamp if TimeStampFormat is UnixTimeStamp. Default value is `MILLISECONDS`. POSSIBLE values are `MILLISECONDS`, `SECONDS`. For example, OneDrive sample Connector sets the value of TimeStampUnit to Seconds using:

  ```
  out_configs.SetTimestampUnit(SECONDS);
  ```

- `void SaaSConfiguration::SetIsLazyInitialization(const bool &value)`

  Use this method to tell SDK whether Metadata Initialization need to be done always at Connection Time OR only when Lazily is required. Possible values are `true, false`. Default value is `false`.

  `true`: Dynamic metadata initialization will be done only when required or only when it is used.

  `false`: Dynamic metadata initialization will be done at Connection time irrespective of whether it will be used or not later in that connection session.

  For example, OneDrive connector sets LazyInitialization to true using

  ```
  out_configs.SetIsLazyInitialization(true);
  ```

- `void SaaSConfiguration::SetIsThrothrottlingSupported (const bool& value)`

  Use this method to tell SDK whether Server supports Throttling. Possible values are `true`, `false`. Default value is `false`.

  SDK handles the Server Throttling if you enable this setting. Forexample, OneDrive Datasource server supports Throttling and hence it sets ThrottlingSupported to true using:

```
out_configs.SetIsThrothrottlingSupported(true);
```

TODO #5: Setup the BrowseConnectMap to be used during Authorization process:

`SaaSAuthBrowseConnectMap` map will contain all the possible values that needed during the whole authorization process with multiple processes. They can be either `required` or `expected` from the `Sequences` under `AuthProfiles`.
Each `Sequences` under `AuthProfiles` represents a step of the authorization process. The required parameters will be used during each step, and each step will generate a result. This result will be used to fill in the expected parameters that are listed under sequences, which may be needed for later authorization process step. `Required` values cab either picked up from the user provided connection string during the initializing process of the map or previous authorization step `expected` values. It also helps you override the name of the connection parameter as you would like to set for your connector.

Define the keys to be used in `SQLBrowseConnect` flow as connection parameters. Some keys for sensitive information or credentials can be used to specify both encrypted value as well as plain text values. In such case, define a separate key to specify encrypted value and separate key to specify the plaintext value. For example, `Auth_AccessToken` is a key that can be used to specify AccessToken in connection string as a plain text and `EncAccessToken` is a key used to specify the encrypted Access Token in connection string.

`SaaSBrowseConnectMap` is used to define both encrypted and plaintext keys to be used in SQLBrowseConnect flow.

The key to specify plaintext value can be defined using `SaaSBrowseConnectMap::SetKey(simba_string&)` method. The key to specify encrypted value can be defined by using `SaaSBrowseConnectMap::SetEncBrowseConnectKey(simba_string&)`.Both of these methods accept string input which is the key you specify to be used in Connection string.

Once defined, it can be added to configuration using a pair:
`out_configs.GetAuthBrowseConnectMap().insert({“KeyName”, SaaSBrowseConnectMap});`

Where “`KeyName`” is one of the `SaaSRequiredParam` you set under a `SaaSAuthSequence` or it can be one of the keys used by SDK to perform OAuth 2.0 authentication. You can also set “`KeyName`” that might be required for your custom authentication to work.

For example,OneDrive Sample Connector sets the following keys in `SaaSAuthBrowseConnectMap` that will be used later to get the required keys and values for each `SaaSAuthSequence` execution for OAuth 2.0 Authentication profile.
`SaaSAuthBrowseConnectMap mapAuthClientId;`

```
mapAuthClientId.SetKey("Auth_Client_Id");
out_configs.GetAuthBrowseConnectMap().insert({"Auth_Client_
Id" , mapAuthClientId});
```

Example: Setting the Auth_Client_Secret and Encrypted Key:

```
SaaSAuthBrowseConnectMap mapAuthClientSecret;

mapAuthClientSecret.SetKey("Auth_Client_Secret");

mapAuthClientSecret.SetEncBrowseConnectKey
("ENCCLIENTSECRET");

out_configs.GetAuthBrowseConnectMap().insert({"Auth_Client_
Secret" , mapAuthClientSecret});
```

TODO #6: Configure Authentication & Parser:

AuthProfiles

Authentication using OAuth 2.0 can be done via REST API as defined by the datasource. It may involve multiple REST API requests to be sent , for example, in case of 3-legged authentication, where one of the API endpoint is used for authorization and the other API endpoint is used to get the AccessToken and RefreshToken (if applicable) by supplying the authorization code in the API request.

While two-legged authentication can be done by using single API Endpoint by supplying all the required fields into the API request and getting the Access Token and Refresh Token (if applicable) in the response.

Access Token can be generated by using the above-mentioned flow (called `GetToken` flow) and it can also be generated by using `RenewToken` flow which will be done when the AccessToken got using `GetToken` flow is expired and need to be renewed using RefreshToken received from `GetToken` flow earlier.

Either you use `GetToken` or `RenewToken` flow, it must go with Sequence of the REST API requests to be sent to server for Authentication. These Sequences can be defined using `SaaSBrowseConnectSequence` class.

Each instance of `SaaSBrowseConnectSequence` corresponds to the

- Authentication URL
- required parameters to be sent.
- request Headers
- HTTP request Type
- Expected Parameters to be received from Server response
- Whether Credentials will go as Basic Authentication header
- Full request Body defined in JSON format with values kept under Placeholders ({{ }}) so that SDK picks up the value of it from the matching Required Parameter set in Connection string at runtime.

Getters and Setters are available in `SaaSBrowseConnectSequence` to Get and Set these properties.

Once the BrowseConnectSequences are defined, add or group them to a particular Authentication sequence with a meaningful name. This can be done by using `SaaSAuthSequence` class. Below are the methods of this class to add SaaSBrowseConnectSequence:

- `SaaSAuthSequence::SetName()` method to set name of a authentication sequence.
- `SaaSAuthSequence::GetSequence()` method to get the reference to vector of `SaaSBrowseConnectSequence` and use insert method on top of it to insert a `SaasBrowseConnectSequence`. You can insert multiple `SaasBrowseConnectSequence` to a single SaaSAuthSequence identified by name set in step 1.

Example of Creating a `SaaSAuthSequence` for GetToken using `SaaSBrowseConnectSequence`:

GetToken Sequence #1:

```
SaaSBrowseConnectSequence browseSeqGetToken1;
{
    SaaSRequiredParam reqParamAuthBaseRedirectUri;
    reqParamAuthBaseRedirectUri.SetKey("Auth_
BaseRedirectUri");
    SaaSRequiredParam reqParamAuthClientId;
    reqParamAuthClientId.SetKey("Auth_Client_Id");
    SaaSRequiredParam reqParamTenantId;
    reqParamTenantId.SetKey("Tenant_Id");
    SaaSRequiredParam reqParamAuthCompletedRedirectUri;
```

```
    reqParamAuthCompletedRedirectUri.SetKey(
    "Auth_CompletedRedirectUri");
    browseSeqGetToken1.GetRequiredParams().push_back
                    reqParamAuthCompletedRedirectUri);
    browseSeqGetToken1.GetRequiredParams().push_back
                          (reqParamAuthBaseRedirectUri);
    browseSeqGetToken1.GetRequiredParams().push_back
                                  (reqParamAuthClientId);
    browseSeqGetToken1.GetRequiredParams().push_back
                                (reqParamTenantId);
    browseSeqGetToken1.SetUrl
("https://login.microsoftonline.com/{{Tenant_
Id}}/oauth2/v2.0/authorize?
scope=User.read&response_type=code&&redirect_uri={{Auth_
BaseRedirectUri}} &client_id={{Auth_Client_ID}}");
browseSeqGetToken1.SetIsAuthorizationRequired(true);
}
```

OneDrive Sample Connector uses three-legged flow for OAuth 2.0 Authentication. So first it needs to Authorize the user to provide access to OneDrive data. The above sequence is created for authorization process where it requires three parameters namely `Auth_Client_ID, Tenant_Id, Auth_BaseRedirectURI` and `Auth_CompletedRedirectURI` which is the full authorization URL got once the authorization process is completed through browser manually.

For now, if AuthorizationRequired is set to true for a particular `SaaSBrowseConnectSequence` then SDK does nothing since this process requires a user interaction via browser. So it is upto user to manually authorize with the server using the URL as specified using `SetURL()` method and once authorization is completed, you get a completed authorization URL which need to be supplied in key "`Auth_CompletedRedirectUri`" in connection string.

Note:

It is important to set this AuthorizationRequired to true for this authorization process sequence otherwise it may result in Unexpected behavior.

Once authorization process is done, we get an authorization code which needs to be supplied into the next HTTP request to get the Access Token. Since "`Auth_CompletedRedirectUri`" is present in the RequiredParams of first sequence, SDK

gets the authorization code from it and saves it in `SaaSAuthBrowseConnectMap` for future use for example, to prepare HTTP requests for subsequent requests.

GetToken Sequence #2:

```
SaaSHeader header1;
header1.SetKey("Content-Type");
header1.SetValue("application/json");
SaaSHeader header2;
header2.SetKey(“Content-Type”);
header2.SetValue(“multipart/form-data; boundary=----------
----------123");
SaaSRequiredParam reqParamAuthClientSecret;
reqParamTenantId.SetKey("Auth_Client_Secret");
SaaSExpectedParam expectedParam1;
expectedParam1.SetKey("Auth_AccessToken");
expectedParam1.SetPath("access_token");
SaaSBrowseConnectSequence browseSeq2;
browseSeq2.GetHeaders().push_back(header1);
browseSeq2.GetHeaders().push_back(header2);
browseSeq2.GetRequiredParams().push_back(reqParamTenantId);
browseSeq2.GetRequiredParams().push_back
(reqParamAuthClientId);
browseSeq2.GetRequiredParams().push_back
(reqParamAuthBaseRedirectUri);
browseSeq2.GetRequiredParams().push_back
(reqParamAuthClientSecret);
browseSeq2.GetExpectedParams().push_back(expectedParam1);
browseSeq2.GetExpectedParams().push_back(expectedParam2);
browseSeq2.GetExpectedParams().push_back(expectedParam3);
browseSeq2.SetIsAuthorizationRequired(false);
browseSeq2.SetUrl(
"https://login.microsoftonline.com/{{Tenant_
Id}}/oauth2/v2.0/token");
browseSeq2.GetHeaders().push_back(header1);
browseSeq2.GetHeaders().push_back(header2);
browseSeq2.SetHttpType(RT_POST);
browseSeq2.SetIsAuthorizationRequired(false);
browseSeq2.SetBody("----------------------------
123\r\nContent-Disposition: form-data; name=\"grant_
type\"\r\n\r\nauthorization_code\r\n-------------------------
---123\r\nContent-Disposition: form-data; name=\"client_
```

```
id\"\r\n\r\n{{Auth_Client_ID}}\r\n--------------------------
-123\r\nContent-Disposition: form-data;
name=\"redirect_uri\"\r\n\r\n{{Auth_BaseRedirectUri}}\r\n----
-----------------------123\r\nContent-Disposition: form-
data; name=\"code\"\r\n\r\n{{Auth_Code}}\r\n----------------
-----------123\r\nContent-Disposition: form-data;
name=\"client_secret\"\r\n\r\n{{Auth_Client_Secret}}\r\n-----
----------------------123--");
```

`SaaSHeader` is used to define Key and Value for HTTP headers in a request. Once defined it can be added using `SaaSAuthBrowseConnectSequence::GetHeader().push_back(SaaSHeader);`

Then we defined the required parameters and the expected parameters for the second sequence. The URL defined using `SetURL()` is the URL to get the Access Token. Request Body is defined using SetBody() method as per OneDrive REST API documentation. The keys for which values are required are kept under curly braces {{ }}.

Both of these GetToken Sequences will be pushed into the SaaSAuthSequence using:

```
SaaSAuthSequence authSeqGetToken;
authSeqGetToken.SetName("GetToken");
authSeqGetToken.GetSequence().push_back(browseSeqGetToken1);
authSeqGetToken.GetSequence().push_back(browseSeq2);
```

The `SaaSAuthSequence` just formed will be pushed into the vector of SaaSAuthSequences.

```
std::vector<SaaSAuthSequence> authSeqOAuth20;
authSeqOAuth20.push_back(authSeqGetToken);
```

Now Insert this Vector of SaaSAuthSequences into the specific authentication profile using :

- `SaaSAuthProfile::GetAuthSequence() ->` to get reference to vector of `SaaSAuthSequence`.
- `Insert`({“`<name>`”, `SaaSAuthSequence`}) -> store pair of authentication typename and it is associated vector of SaaSAuthSequence using Insert method of vector got from Step 1.

Example:

```
authProfile.GetAuthSequence().insert({ "OAuth 2.0",
authSeqOAuth20 });
```

Use `SaaSAuthProfile::GetTypes()` method to get the reference to the vector of strings representing the authentication types supported by this `SaaSAuthProfile` and use push_back method to insert the authentication typename.

Example:

```
authProfile.GetTypes().push_back("OAuth 2.0");
```

SaaSAuthProfile Methods:

- `SaaSAuthProfile::SetTokenType()`:

  This method is used to set the Token type to be set in the HTTP request. The possible values are: "`Bearer`".

- `SaaSAuthProfile::.SetIsExpirationDataAvailable()`:

  Set it to true if expiry time is available along with the AccessToken and RefreshToken.

- `SaaSAuthProfile::setIsAutoRefreshSupported()`:

  Set it to true if server supports Auto Refresh of AccessToken using Refresh token in case AccessToken is expired.

- `SaaSAuthProfile::SetRefreshTokenWithinRange()`:

  This flag indicates whether Token is refreshed before or after expiration date and time.

- `SaaSAuthProfile::SetVerifyHost()`

  This flag indicates whether Host name should be verified against the name in Server's SSL Certificate.

- `SaaSAuthProfile::SetVerifyPeer()`

  This flag indicates whether to verify the Peer's SSL certificate.

Once the `SaaSAuthProfile` is created, it can be added to the configuration using: `SaaSConfiguration::SetAuthProfiles(SaaSAuthProfile)`.

AuthenticationHandler

To support one of the SDK supported Authentication types in your Connector, simply create an Authentication Handler for the desired Authentication type using Authentication Factory class. Authentication Factory class allows you to create an

Authentication handler for any of the Authentication types supported by default by SDK.

```
IAuthenticationHandler*
SaaSAuthenticationFactory::CreateAuthenticationHandler(simba_
string auth_type);
```

Where auth_type is the name of authentication type and can be one of the following values:

```
SAAS_AUTH_VALUE_OAUTH_2
SAAS_AUTH_VALUE_BASIC
SAAS_AUTH_VALUE_ACCESS_TOKEN
```

Once you have an instance of "IAuthenticationHandler", it can be added into Configuration using

```
void SaaSConfiguration::AddAuthHandler(
                const Simba::Support::simba_string& in_
authName,
                IAuthenticationHandler * in_authHandler);
```

Where

`in_authName` is the name of the authentication type your connector will accept as input in the Auth_Type parameter in the connection string.

`In_authHandler` is the corresponding instance of `IAuthenticationHandler` for the `in_authName`.

Example: Create and Register AuthenticationHandler for OAuth 2.0:

```
{
    IAuthenticationHandler* authHandler =
    SaaSAuthenticationFactory::CreateAuthenticationHandler
                                (SAAS_AUTH_VALUE_OAUTH_2);
    out_configs.AddAuthHandler("OAuth_2.0", authHandler);

    out_configs.AddAuthHandler("OAuth 2.0", authHandler);
}
```

Sometimes the HTTP request formed by SDK for Authentication might not be in the way you want it to be, in such cases you can implement your own HTTPRequestModifier by extending the `IHTTPRequestModifier` class. For more details check section *Modify HTTP Request and Response>*.

You need to implement a Custom Authentication handler for Authentication types not supported by default by SDK. Refer Section *Authentication* under *Custom Extensions*.

Parsers

Parser is used to Parse the REST API response data. SDK has predefined parser available to use for JSON, XML and CSV data. To use these parsers, you need to register a parserhandler for a specific type.

To register the ParserHandler for the specific type use the method:

```
IParserHandler* CreateParserHandler(const simba_string& in_
parserType);
Where
In_parserType accepts string and can be one of these values:
SAAS_CSV_PARSER
SAAS_XML_PARSER
SAAS_JSON_PARSER
```

This returns an instance of `IParserHandler` which can be used to set ParserHandler for a specific content type in the `SaaSConfiguration`.

To set ParserHandler for a specific HTTP response type, use:

```
void SaaSConfiguration::AddParserHandler(
             const Simba::Support::simba_string& in_
contentType,
              IParserHandler * in_parserHandler);
```

Where:

`In_contentType` is one of the HTTP response types for example, application or json, application or xml.

`In_parserHandler` is the handler to be used for in_contentType.

Example: Create and Register a Parser handler for JSON ResponseType:

```
{
    SaaSParserFactory factory;
    IParserHandler* parserHandler =
    factory.CreateParserHandler(SAAS_JSON_PARSER);
    out_configs.AddParserHandler("application/json"
                                 ,parserHandler);
}
```

You need to create a Custom Parser handler to implement Parser handling for Response types not supported by SDK. Refer Section *Parsers* underCustom Extensions.

TODO #7: Configure Tables & Columns:

`Configuration.cpp` file is a big file since Most of the Driver Configurations are configured in this file. And hence we have divided the configurations into separate .cpp files so that this main file is more readable and easier to maintain.

Each table is configured this way:

- Create a user defined function with name as table name and declare it in `ConfigurationHelpers.h`.
- Define the function in `TableName.cpp` file.
- Include `ConfigurationHelpers.h` in your main file `Configuration.cpp` and then call the tablefFunction with their name, for example, TableName() to configure the table.

Important things to note about tables and columns:

- A table may represent a REST API Object. For example, If Datasource expose a REST API to fetch the list of orders or to fetch a particular order with an order Id, here Orders is considered as REST Object, table in your Custom Connector.
- The properties of a REST API Object, OrderId, Grand Total, Order Items can be considered as columns for that particular table Orders. Thus, columns represent the REST API object properties which we get in the REST API response in some format: JSON, XML, CSV.
- For each table, we configure the REST APIs to be used for that table to fetch data through REST API and configure each column to refer to some particular property of a REST API object in the API response.
- The REST API response may look like it has nested sub REST API objects for example, list of items for a particular order then we can also create a so called sub table for it with named OrderItems which represents the items for each particular order. A sub table thus created is called a virtual table. A single main table or parent table can have multiple virtual tables or child tables which represent the sub objects and it's properties. This means that both virtual table and it's parent or main table share the same REST API request to fetch data but represent different data that is related with the parent or main table's key column.

  For Instance, the key columns for Orders table could be OrderId which is referred in the sub table OrderItems to relate a list of items with a particular OrderId. Thus OrderId is a foreign key in the OrderItems table and primary key in the Orders table.

The above-mentioned points can be served as a brief summary to give you an idea of how the tables and columns will be configured in the next steps.

Have a look at the OneDrive REST API here:

https://learn.microsoft.com/en-us/onedrive/developer/rest-api/api/driveitem_get?view=odsp-graph-online#request

It presents an API to get the OneDrive root folder information. An example request and JSON response is also present. We can support this as a table with name DriveRoot which gives us the data of OneDrive Root folder. Let us design or configure this table.

Step 1: Declare a Function with name as your Tablename in "`ConfigurationHelpers.h`" file.

```
void DriveRoot(Simba::SaaSSDK::SaaSConfiguration& io_configs);
```

Step 2: Create a new (.cpp) file with name as your tablename and define the function "DriveRoot" in that new file.

Include 2 files "`ConfigurationHelpers`.h" to follow the declaration of function with it's definition and "`SaaSConfiguration.h`" to configure the table.

`SaaSTable` represents a single table. It has the following properties to be set for table:

- `void SetTableName(const Simba::Support::simba_wstring& in_tableName):`

  Set Table name using this method. Pass the table name as parameter.

- `void SetTableSchemaName(const Simba::Support::simba_wstring& in_tableSchemaName):`

  Set Table schema name using this method. Pass Schema name as parameter.

- `void SetTableCatalogName(const Simba::Support::simba_wstring& in_catalogName);`

  Set Table Catalog name using this method. Pass Catalog name as parameter.

- `void SetSortable():`

  Sets true to represent that the table's REST APIs supports sorted rows to be returned from the Data source itself. If API doesn't support then this function need not be called.

- `void SetPageable()`:

  Sets if the table data needs to be fetched in mulitple pages. If Not then this function should not be called.

- `void AddColumn(SaaSTableColumn)`:

  Use this method to add a Column to Table.

- `void SetPkeyColumns(const SaaSTablePKeyColumn& in_pkeyColumns)`:

  Sets the Primary Key Columns for a table.

- `void AddForeignKeyColumn(SaaSFKeyColumn& in_fKeyColumns)`:

  Add a Foreign Key column for a table.

- `void SetAPIAccess(const SaaSTableApiAccess& in_apiAccess)`:

  Sets the table's access details. On how to read the data for the table from the Data Source.

`SaaSTableColumn` represents a single column. The methods available in this class are:

- `void SetName(const Simba::Support::simba_wstring& value)`:

  Set the name of the column. Pass the column name as a parameter.

- `void SetMetadata(const SaaSMetadata& value)`:

  A column has a metadata like SQL DataType, Size. This metadata can be set using SaaSMetadata and can be added into column using this method.

- `void SetNullable(const bool& value)`:

  Indicates whether the column can contain NULL values if no data is fetched from server for this particular column.

- `Void SetUpdatable(const bool&)`:

  This is used to make column writable for example, Update or Delete or Create new records in a table using this column. This should always be false for all the columns since Simba REST SDK is a read only SDK and supports only READ operations.

- `void SetPassdownable(const bool&):`

  Whether the filters or sort applied to a select query for this particular table on this particular column can be Passdownable to the actual REST API request so that we get the filtered or sorted result back from server itself.

- `void SetSvcRespAttrListResult(const Simba::Support::simba_wstring&):`

  The path to the REST API object property in the List Endpoint Response (i.e. Multiple items in same response) of a REST API which this particular column refers to get data from.

- `void SetSvcRespAttrItemResult(const Simba::Support::simba_wstring&):`

  The path to the REST API object property in the Item Endpoint Response (i.e.Single item in a response) of a REST API which this particular column refers to get data from.

- `void SetSvcReqParamQueryMapping(const Simba::Support::simba_wstring&):`

  The mapping to use when using this column is used in the query of the URL.

- `void SetColumnPushDownMapping(const Simba::Support::simba_string&):`

  The mapping to use when passing column as query parameter in the URL.

- `void SetFilteringSupportedWithItemEndPoint(bool):`

  Flag to indicate if the column is supported with the item endpoint. (default = true).

SaaSMetadata:

It is used to define metadata for a single column. Following are it's properties:

- `void SetSourceType(SourceType):`

  Data type from the DataSource for this column.
  Possible values are: `UNKNOWN, INTEGER, NUMBER, STRING, ARRAY, OBJECT, BOOLEAN, JNULL`

- `void SetSqlType(const simba_int16 &value):`

  Set the SQL DataType.
  Possible values are: `SQL_VARCHAR, SQL_WVARCHAR, SQL_`

```
LONGVARCHAR, SQLWLONGVARCHAR, SQL_CHAR, SQL_WCHAR, SQL_
DECIMAL, SQL_FLOAT, SQL_DOUBL, SQL_INTEGER, SQL_BIGIN,
SQL_NUMERIC, SQL_BIT, SQL_LONGVARBINARY, SQL_TIMESTAMP,
SQL_DATETIME, SQL_DATE
```

- `void SetLength(simba_int32)`

  Set the data type length to be supported.

- `void SetIsUnsigned(bool value):`

  To tell if the value for numeric type is signed or unsigned.

- `void SetScale(const simba_int64&):`

  Set scale for SQL Type.

- `void SetPrecision(const simba_int64&)`

  Set precision for SQL Type.

SaaSColumnPushdown:

Column pushdown enables your Custom Connector to Get Data for only the required set of columns from DataSource via REST API. Thus, it reduces the response body of server response in Bytes and hence the response time also, avoiding the extra information not needed to be fetched from server.

You can enable the column pushdown support for a table using:

```
SaaSColumnPushdown::SetSupport(bool);
```

By default, the support is disabled, to enable it use the above method and call:

```
SaaSColumnPushdown::SetSupport(true);
```

Below are the additional methods of SaaSColumnPushdown to configure column pushdown:

- `void SetParameterFormat(ParameterFormat):`

  Set the parameter format. Possible values are:

  `PARAM_FORMAT_URL`: The parameter is embedded into URL.

  `PARAM_FORMAT_BODY`: The parameter is a part of API Request Body.

  `PARAM_FORMAT_QUERY`: The parameter is a URI parameter (Default).

- `void SetSvcReqParamKey(const Simba::Support::simba_string &):`

  Set the Query parameter key name required to be appended into request URL.

- `void SetQueryParam(const Simba::Support::simba_string&):`

  A delimited query parameter values required to be appended into request URL as default.

- `void SetSvcReqParamDelimiter(const Simba::Support::simba_string&):`

  Delimiter between columns in the request URL. Default is ",".

Example: Configure ColumnPushdown:

```
    SaaSColumnPushdown columnPushdown;
    columnPushdown.SetSupport(true);
    columnPushdown.SetParameterFormat(PARAM_FORMAT_QUERY);
    columnPushdown.SetSvcReqParamKey("fields");
    columnPushdown.SetSvcReqParamDelimiter(",");
    columnPushdown.SetQueryParam("orderId, ordername,
items.id, items.name, grandTotal");
```

Let us understand this with our classic example of orders table.

Orders API might give more data in the API response which might not be needed as per our use case. So we only request specific data we need for example orderId, ordername, items.id, items.name, grandTotal. This is the list that will be appended into the API Request as a Query parameter since ParameterFormat is set to PARAM_FORMAT_QUERY. If the REST API to get Orders is v1 or orders then the final request looks like this:

https://{HOST}/v1/orders?fields=orderId,ordername,items.id,items.name,grandTotal

where {HOST} is the hostname configured for your Datasource.

In the API response, data for only the fields we requested will be present.

Important things to note:

- If you have enabled Column Pushdown and kept SetQueryParam() to empty string which is it's default value, then for which all columns to request data will be decided at run time. It will be determined by for how many columns you requested data in your SQL Query.

- The details like whether column pushdown can be enabled? Or what will be the Query parameter key for it? You need to check this in REST API documentation of the DataSouce for which you are building your Custom Connector.

Once you have defined the Column Pushdown for your table, it can be added into table using:

```
void SaaSTable::SetColumnPushdown(const SaaSColumnPushdown&
in_columnPushdown)
```

```
Eg: saasTable.SetColumnPushdown(columnPushdown);
```

SaaSPk:

Used to define a primary key column for a table. The following methods are used:

- `void SaaSPk::SetPkColumn(const unsigned int &)`:

  Provide the column index (starts from 0 for first column) to specify the column as a primary key.
- `void SaaSPk::SetSequenceNumber(const unsigned int)`:

  Sequence number assigned to a column within a Primary Key in case multiple columns are used to form a primary key.

Generally, to avoid calculating the column index number, you can define the Primary Key immediately after defining a Primary Key column using SaaSTableColumn by using this logic:

```
{
  // Column definition which will be defined as a primary
key.
  SaaSTableColumn column;

}

{
  // Set the recently defined column as a primary key.
  SaaSPK pk_table;
  pk_table.SetPkColumn(table.GetColumns().size() - 1);

}
```

SaaSTablePKeyColumn:

This class is used to set the primary key column indexes, primary key column names and sequence numbers under a specific group identified by a name called Primary

Key Name.

Following are the methods:

- `void SaaSTablePKeyColumn::SetPKeyName(const Simba::Support::simba_wstring&)`:

  Set the custom primary key name.

- `void SaaSTablePKeyColumn::AddPK(SaaSPK&)`:

  Add SaaSPk column.

- `void SaaSTablePKeyColumn::AddPKColumnNames(const Simba::Support::simba_wstring&)`:

  Add the name of a Primary Key Column name.

Once the `SaaSTablePKeyColumn` is defined for a table it can be added into `SaaSTable` using:

```
void SaaSTable::SetPkeyColumns(const SaaSTablePKeyColumn&)
```

SaaSForeignKeyColumns:

Define a single Foreign Key column using this class. Below are the methods to be used:

- `void SaaSForeignKeyColumns::SetFKeyColumnName(const Simba::Support::simba_string &)`:Set the Foreign Key column name for current table.
- `void SetPrimaryKeyColumnName(const Simba::Support::simba_string &)`:Set the Primary Key Column name which is being referenced from the Referenced Table to this current table.

SaaSFKeyColumn:

Define Foreign Key columns for table using this class. Below are the methods to be used:

- `void SaaSFKeyColumn::SetForeignKeyColumns(const Simba::SaaSSDK::SaaSForeignKeyColumns &)`:Set the Foreign Key columns created using SaaSForeignKeyColumns.
- `void SaaSFKeyColumn::SetReferenceTable(const Simba::Support::simba_string &)`:Set the reference Table name from where the ForeignKey column is being referenced.

- `void SaaSFKeyColumn::SetReferenceTableSchema(const Simba::Support::simba_string &)`:Set the Schema name of a reference table from where the ForeignKey column is being referenced.

Once the `SaaSFKeyColumn` is defined, it can be added into the `SaaSTable` using below method:

```
void SaaSTable::AddForeignKeyColumn(SaaSFKeyColumn& in_
fKeyColumns);
```

Example: Set ForeignKey column

If we take a classic example of Orders table and its sub table called Items (also called Virtual Table) then the foreign key relationship can be defined as follows in Items table assuming `Order_Id` is a primary key column in Orders table and the same column referenced in Items table is with name `Item_Order_Id`.

```
{
    SaaSForeignKeyColumns foreignKeyColumn;
    foreignKeyColumn.SetFKeyColumnName("Item_Order_Id");
    foreignKeyColumn.SetPrimaryKeyColumnName("Order_Id");
    SaaSFKeyColumn fKeyColumn;
    fKeyColumn.SetForeignKeyColumns(foreignKeyColumn);
    fKeyColumn.SetReferenceTable("Orders");
    saasTable.AddForeignKeyColumn(fKeyColumn);
}
```

Assuming that there is only one schema used, so setting the reference table schema is not required.

SaaSReadAPIEndpoint:

This class is used to specify the REST API Endpoint details for a table which will be used to fetch the list of records, a single record as well as any prerequisite data that need to be fetched before fetching the actual data from server.

It has the following methods:

- `void SaaSReadAPIEndpoint::SetListEndPoint(const Simba::Support::simba_string &)`:Set the URI endpoint to connect to this resource when obtaining a list of item.
- `void SaaSReadAPIEndpoint::SetItemEndPoint(const Simba::Support::simba_string &)`:Set the URI endpoint to connect to this resource when obtaining a single item. Use an Item Id in curly braces {{}} to

denote the location of the primary key in the endpoint.

- `void SetType(const Simba::Support::simba_string &):`

  This field describes how to form the correct endpoint(s) to get the full list of items.

  Possible values are:
  "`ENDPOINT_ONLY`": Endpoint doesnot require any prerequisite data to execute.

  "`PREREQ_CALLS`": Endpoint requires some prerequisite data to be feed into before execution which can be fetched from prerequisite Endpoint.

- `void SetPreReqCall(SaaSPreReqCall&)`:Set the PreReqCall for current API Endpoint which will execute prior to the execution of the Actual API Endpoint. PreReqCall Endpoint is used to fetch required data to be feed into the Actual API Endpoint Request. For more details, please look at PreReqCall section.
- `void SetPaginationData(SharedPtr<ISaaSPaginationData> value):`

  If data is received over multiple pages of API requests, then pagination is required to be configured. This method is used to set the pagination parameters like Size of each Page, Identifier for Next page, Identifier for End of Pagination etc.
  Pagination data will be different for each pagination type. ISaaSPaginationData is an Interface which is extended by each pagination type supported by SDK, for example, Index Based and Response based to add pagination data for each pagination type. So you need to manually set the pagination data as per your Datasource documentation using one of the classes: SaaSIndexBasedPaginationData or SaaSResponseBasedPaginationData based on which pagination type you use.

  Please look at Pagination section for more details.

SaaSReadAPI:

This is used to define the REST API structure for a particular table. It has following methods:

- `void SaaSReadAPI::SetMethod(RequestType):`

  Set the HTTP Request Method. Possible values are:

  RT_GET: Use HTTP GET method (Default)

  RT_POST: Use HTTP POST method

RT_NONE: No HTTP method specified

- `void SaaSReadAPI::SetBodySkeleton(const Simba::Support::simba_wstring&):`

  Set the skeleton of the payload's body. When building a payload body to send to the server.

- `void SaaSReadAPI::SetEndPoint(const Simba::SaaSSDK::SaaSReadApiEndpoint &):`

  Set the ReadAPIEndpoint which has the Endpoint details containing information on how to retrieve items from the data source server.

- `void SaaSReadAPI::SetAccept(const Simba::Support::simba_string &):`

  Set the HTTP Accept header for the Read Request.

- `void SaaSReadAPI::SetContentType(const Simba::Support::simba_string &):`

  Set the HTTP Content type header for the Read Request.

- `void SaaSReadAPI::SetListRoot(const Simba::Support::simba_wstring &):`

  If the row data is under common root path then that common root path can be set as a list root. If the list root is set the paths should not contain it. For example, for below JSON.

```
{
                "results" : [
                    { "a" : "aval1" , "b" : "bval1" },
                    { "a" : "aval2" , "b" : "bval2" },
                    { "a" : "aval3" , "b" : "bval3" },
                    { "a" : "aval4" , "b" : "bval4" }
                ]
}
```

In the above JSON, we have two columns (i.e. “a” and “b”) and each row is represented as a JSON object. All rows have a common path ,results, so if you set the list root as results then column paths only need to set the rest of the path to get data, a and b, and Not results.a and results.b.

- `void SaaSReadAPI::SetItemRoot(const Simba::Support::simba_wstring &)`:

  Same concept as ListRoot. Just that this is used to set the root path for Item Endpoint.

- `void SaaSReadAPI::SetParameterFormat(const ParameterFormat&)`:

  Set the parameter format if any parameter required for API request. Possible values are:

  `PARAM_FORMAT_URL`: The parameter is embedded into URL.

  `PARAM_FORMAT_BODY`: The parameter is a part of API Request Body.

  `PARAM_FORMAT_QUERY`: The parameter is a URI parameter (Default).

- `void SaaSReadAPI::SetPaginationHandler (IPaginationHandler* in_handler)`:

  Set the Pagination handler for this particular READ API.

- `Void SaaSReadAPI::SetHttpReqModifierHandler (IHttpRequestModiferHandler*)`:

  In case the API Request formed by SDK needs to be customized before sending it to Server, then use this method to Register the IHTTPRequestModiferHandler object you created.

- `void SaaSReadAPI::SetHttpRespModifierHandler (IHttpResponseModiferHandler*)`:

  In case the API Response received from the server needs to be customized before parsing the response to fetch data out of it, then use this method to register the IHttpResponseModiferHandler object you created.

SaaSTableAPIAccess:

This class is used to specify the API Access for the current table. The methods of this class are as follows:

- `void SaaSTableAPIAccess::SetReadApi(const SaaSReadApi &)`:

  Set the Read API Access created using SaaSReadAPI.

Once the Read API is set using above method, SaaSTableAPIAccess can be set to a particular table using the below method:

```
void SaaSTable::SetAPIAccess(const SaaSTableApiAccess&)
```

Following the above guidelines, the Table “DriveRoot” is created as mentioned below for OneDrive Sample RESTful connector:

Pagination

To support Pagination for table, follow these steps:

- If you use SDK defined Pagination type, then create a Pagination Handler using inbuilt method of `SaaSPaginationFactory` class. Or define your own Pagination Handler by extending the `IPaginationHandler` class.
- Use `SaaSReadApi::SetPaginationHandler(IPaginationHandler)` to set the pagination handler created in step 1.
- Define the pagination data page size, next page identifier, end of pagination identifier. If you are using one of the SDK defined pagination types, then either use `SaaSIndexBasedPaginationData` or `SaaSResponseBasedPaginationData` class to define the pagination data. Or if you are using a Custom Pagination type then create a class by extending the `ISaaSPaginationData` interface and implement all the methods that are required to set and get pagination data and then use that custom class to define the pagination data.
- Set the Pagination Data for SaaSReadAPIEndpoint using method:

  ```
  SaaSReadAPIEndpoint::SetPaginationData
  (SharedPtr<ISaaSPaginationData>);
  ```

Let us go through each step using an example:

- Create Pagination Handler (SDK Defined one):

  In this example, we will use Index Based Pagination so, will create a handler using:
  ```
  IPaginationHandler* paginationHandler =
  SaaSPaginationFactory::CreatePaginationHandler(SAAS_PAGE_
  TYPE_INDEX_BASED);
  ```

  You can also pass the following values to CreatePaginationHandler based on your requirement:

  `SAAS_PAGE_TYPE_HEADER_BASED` - Used when the Pagination data is present in response headers.

`SAAS_PAGE_TYPE_TOKEN_BASED` - Used when there is Token present for the next page in the API response.

`SAAS_PAGE_TYPE_BODY_BASED` - Used when the Next page URL is present in the Response body.

- Set Pagination Handler to `SaaSReadApi`.
  Once you define the `SaaSReadApi` for your table, set Pagination handler for it using:

  ```
  SaaSReadApi::SetPaginationHandler(paginationHandler);
  ```
  Where `paginationHandler` is the handler created in step 1.

- Define Pagination data for Index based Pagination:

  `SaaSIndexBasedPaginationData` class has following properties to be set to define pagination data:

  - `void SetMaxPageSize(const simba_int64&):`

Max page size refers to the maximum number of records that will be fetched in a single page.

- `void SetPageStartIndex(const simba_int64&):`

Since there are multiple pages, each page is identified by an index number sequentially. Set the start index using this method.

- `void SetOffsetType(OffsetType);`

OffsetType defines how the Offset is calculated for the next page in sequence. If it is by the number of items in the current page or it is by the current page number. Possible values are: `OFFSET_ITEM`, `OFFSET_PAGE`, `OFFSET_NONE` (No offset at all).

- `void SetReqQueryOffSetKey(const`

  `Simba::Support::simba_string&);`

The query parameter key to be set in the API request to specify the offset value.

- `void SetReqQueryLimitKey(const`

  `Simba::Support::simba_string&):`

The query parameter key to be set in the API request to specify the maximum number of items to be returned.

- `void SetTerminationType(TerminationType):`

Specifies how the termination of pagination is determined. Possible values are:

`TYPE_ROWCOUNT` - Total number of rows available for the API.

`TYPE_PAGECOUNT` - Total number of pages, for the row count sent. Row count if set once should not be changed next else it will cause in miscalculation of number of pages required.

`TYPE_EMPTYPAGE` - Last page number not available

`TYPE_NEXTPAGEURL` - Next page URL fetched from the response. Terminate when it is empty or not found.

`TYPE_TOKEN` - Token to be used to form the next page URL.Terminate when it is empty or not found.

- `void SetTerminationKeyPresentIn (TerminationKeyPresentIn):`

Where the Termination Key is present? Possible values are:

`KEY_HEADER` - The key value are present in the http headers.

`KEY_BODY` - The key value are present in the http response body.

`KEY_NONE` - Key value is not available, can be used for `TYPE_EMPTYPAGE`.

- `void SetRespTerminationElement(const`

  `Simba::Support::simba_string&);`

Set the termination element if present in the response body. Path to termination key in the response, it should be a (.) dot separated path.

Example: Set the Pagination Data:

```
SharedPtr<SaaSIndexBasedPaginationData> paginationInfo(new
SaaSIndexBasedPaginationData());
    {
       paginationInfo->SetMaxPageSize(100);
       paginationInfo->SetPageStartIndex(1);
       paginationInfo->SetOffsetType(OFFSET_PAGE);
       paginationInfo->SetReqQueryOffSetKey("page");
       paginationInfo->SetReqQueryLimitKey("count");
       paginationInfo->SetTerminationType(TYPE_ROWCOUNT);
```

```
        paginationInfo->SetTerminationKeyPresentIn(
                                                KEY_BODY);
        paginationInfo->SetRespTerminationElement("total");
    }
```

- Now Set the Pagination Data in SaaSReadApiEndpoint using:

  `SaaSReadAPIEndpoint::SetPaginationData(paginationInfo);`

  Where `paginationInfo` is PaginationData defined in above step.

Table Implementation Example from Sample Driver

```
void OneDriveDriveRoot(SaaSConfiguration& io_configs)
{
    SaaSTable table;
    table.SetTableCatalogName(L"OneDrive");
    table.SetTableName(L"DriveRoot");
    table.SetTableSchemaName(L"OneDrive");
       // Primary Key
    {
        SaaSTablePKeyColumn tablePKey;
        tablePKey.SetPKeyName("pk_DriveRoot");
        table.SetPkeyColumns(tablePKey);
    }
    // Foreign Keys
    // Static Columns
    {
       {
            SaaSTableColumn column_table;
            column_table.SetName("@odata.context");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult
("@odatacontext");
             table.AddColumn(column_table);
        }
```

```
        {
            SaaSTableColumn column_table;
            column_table.SetName("createdDateTime");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult
("createdDateTime");
            table.AddColumn(column_table);
        }
        {
            SaaSTableColumn column_table;
            column_table.SetName("id");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult("id");
            table.AddColumn(column_table);
        }
        {
            SaaSTableColumn column_table;
            column_table.SetName("lastModifiedDateTime");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult
("lastModifiedDateTime");
            table.AddColumn(column_table);
        }
```

```
        {
            SaaSTableColumn column_table;
            column_table.SetName("name");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult("name");
            table.AddColumn(column_table);
        }
        {
            SaaSTableColumn column_table;
            column_table.SetName("webUrl");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult("webUrl");
            table.AddColumn(column_table);
        }
        {
            SaaSTableColumn column_table;
            column_table.SetName("size");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_INTEGER);
            metadata_column_table.SetIsUnsigned(true);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult("size");
            table.AddColumn(column_table);
        }
        {
            SaaSTableColumn column_table;
```

```
            column_table.SetName("parentReference_driveId");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult(
                                  "parentReference.driveId");
            table.AddColumn(column_table);
        }
        {
            SaaSTableColumn column_table;
            column_table.SetName("parentReference_
driveType");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult(
                                 "parentReference.driveType");
            table.AddColumn(column_table);
        }
        {
            SaaSTableColumn column_table;
            column_table.SetName("createdDateTime");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult(
                               "fileSystemInfocreatedDateTime");
            table.AddColumn(column_table);
        }
```

```
        {
            SaaSTableColumn column_table;
            column_table.SetName("fileSystemInfo_
lastModifiedDateTime");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_VARCHAR);
            metadata_column_table.SetLength(8192);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult(
                      "fileSystemInfolastModifiedDateTime");
            table.AddColumn(column_table);
        }
        {
            SaaSTableColumn column_table;
            column_table.SetName("folder_childCount");
            SaaSMetadata metadata_column_table;
            metadata_column_table.SetSqlType(SQL_INTEGER);
            metadata_column_table.SetIsUnsigned(true);
            column_table.SetMetadata(metadata_column_table);
            column_table.SetNullable(false);
            column_table.SetUpdatable(false);
            column_table.SetPassdownable(false);
            column_table.SetSvcRespAttrListResult(
                                    "folderchildCount");
            table.AddColumn(column_table);
        }
    }
// Skeleton Columns
{
}
    // APIAccess
    {
        SaaSTableApiAccess table_apiAccess;
        // ReadAPI
        {
            SaaSReadApi table_readApi;
            // ReadAPI Endpoints
            {
```

```
                SaaSReadApiEndpoint table_readApiEndpoint;
                table_readApiEndpoint.SetListEndPoint(
                                            "/me/drive/root");
                 table_readApiEndpoint.SetType("ENDPOINT_
ONLY");
                 table_readApi.SetEndPoint(table_
readApiEndpoint);
             }
             table_readApi.SetMethod(RT_GET);
             table_readApi.SetAccept("application/json");
             table_readApi.SetContentType("application/json");
             table_readApi.SetParameterFormat(PARAM_FORMAT_
BODY);
             table_apiAccess.SetReadApi(table_readApi);
         }
         table.SetAPIAccess(table_apiAccess);
    }
    io_configs.AddTable(table);
}
```

Passdown

REST API endpoints supports filtering, sorting of data by passing the required parameters to filter or sort data. To perform column based filtering or sorting, a column can be passed down to an endpoint to get the filtered or sorted data back from server. If the server doesn't support filtering or sorting using some column, then column is not passed down and filter or sort is performed by Simba Engine SDK, and we get filtered or sorted result in output.

We should leverage this feature for most of the API Endpoints (if it is supported by data source) to increase the connector's performance since we get the filtered or sorted result directly from API response, rather than fetching all data first and then let Simba Engine SDK filter the result for us and then we get the filtered result.

To enable passdown feature for a column , enable filtering or sorting using a column via REST API and use the `SetPassdownable(const bool)` method of `SaaSTableColumn` to Set Passdownable to true for column.

In case we request a single resource or item to be fetched, for example, SQL SELECT query with Where clause specifying the Id value of item or resource to be fetched, the item endpoint should be used which requires only one request to be sent to server to get data of just single requested item. It also requires to passdown the value of the item that is specified in SELECT query. If any column's value need to be added into itemendpoint at run time then it can be specified using `AddItemEndpointColumnNames(simba_string)` method of `SaaSTable`.

For example:

SQL query with where clause:

```
SELECT * from <someTable> WHERE Id = <someValue>
```

And the item endpoint to be used is

```
/rest/v1/items/{Id}
```

Here the value of Id specified in the where clause of Select query can be passed down to item endpoint to replace the {id} placeholder with the actual value of Id supplied in query. It will result in sending only one request to server to get data for single item.

Some important points for Passdown:

CQE (Collaborative Query Execution) is a way to execute the queries in which SDK needs data source specific information from Driver for example, Filter or Sort parameter keys, Operations supported, Passdownable columns and SDK uses this information to build a tree structure to execute the query with passdown. To use

pasdown feature of SDK, you must define the CQE information using `SaaSCQEFilterInfo` or `SaaSCQESortInfo`.

Example (Set CQE Sort info. for a single table):

bool opSupportMap[SAAS_OP_SORT_END] = {};

```
{
// Build CQE Sort Info.
SaaSCQESortInfo sortInfo;
sortInfo.SetIsDirectionFirst(false);
sortInfo.SetIsMultipleParamSupported(false);
sortInfo.SetOpString(SAAS_OP_SORT_DESC, " DESC");
sortInfo.SetOpString(SAAS_OP_SORT_ASC, " ASC");
sortInfo.SetSortQueryKey("orderBy");
// Set CQE Sort Info.
table.SetCQESortFrom(CQE_SORT_FROM_TABLE);
table.SetCQESortInfo(sortInfo);
}
```

For more information, check both classes in *Simba REST SDK C++ API reference*.

Any column for which filtering is to be supported with list endpoint but not an ItemEndpoint then let the SDK know about it using method `SetFilteringSupportedWithItemEndpoint(bool)` of `SaaSTableColumn`. Pass false value in this case. Default is true.

If filtering is supported using ItemEndpoint, then all the columns that are supported to filter using ItemEndpoint should be added using method `AddItemEndpointColumnNames(simba_string)` method of `SaaSTable`.

A column should be set to passdown by using `SetPassdownable(bool)` method of `SaaSTableColumn` if filtering or sorting is to be supported for it. Default is false.

Also supply the list of operators supported to filter or sort data for a particular column using method void `SetOperationSupport(simba_int32 in_op, bool in_supported)` of `SaaSTableColumn`. For a list of available operators that can be set, see the `Simba::SaaSSDK::SAAS_COMP_*` and `Simba::SaaSSDK::SAAS_OP_*` variables in *Simba REST SDK C++ API reference*.

PreReqCalls

A call to an endpoint requires some prerequisite data to be feed into it before final execution to get the rows of data. In this case the endpoint which is used to fetch the prerequisite data is termed `PreReqCall`. `PreReqCall` request's response is examined, and the values present in the data path is picked up and the actual endpoint request is formed.

Consider an example of getting the details of items ordered today. We have the following endpoints available to get the data:

`/rest/v1/orders ->` To get the list of orders (Prerequisite Endpoint).

`/rest/v1/items/{id}` -> To get detail of a single Item.

This operation can be done through two steps:

- Get the Id of items ordered today by using the below `PreReqCall` endpoint:

  `/rest/v1/orders?sinceDate=TODAY` where TODAY is today's date and `sinceDate` is the query parameter used to filter the orders by order date.

  In the response of this API request we will get the IDs of items ordered today which will be used in next step to get the item details.
- The previous endpoint just gives the item ID rather than the full details of an item. To get the full details of an ordered item, the below endpoint needs to be used:

  `/rest/v1/items/{id}` where `{id}` will be replaced by the item Id got from previous step. This endpoint will be called N ,number of times to get the details of each item where N is the total number of unique item IDs retrieved from Step one.

To support prerequisite data fetch, use `SaaSPreReqCall` class to configure your prerequisite call structure and the values to read from response to be feed into your final REST API request or endpoint configured under `SaaSReadApiEndpoint`. The endpoint type must be set to "`PREREQ_CALLS`" to support data prefetch.

SaaSSvcReqParamKey

Used to specify the data to be fetched from prereqcall response and it's path. Also used to specify, if the data is to be sent as a query parameter, is referenced field (a referenced field is cached by SDK and later when required will be used from cache), max values per call for example,how many values will be processed by the READ API Endpoint at a time in single request. Default is one.

SaaSSvcReqParamKey methods:

- `void SaaSSvcReqParamKey::SetKeyName(const Simba::Support::simba_string& value)`

  Set the key name to be used to refer to prerequisite data. This key name will be used in the actual endpoint to refer to this prerequisite data.
- `void SaaSSvcReqParamKey::SetSvcRespAttrField(const Simba::Support::simba_wstring& value)`

  Set the path to the prerequisite data in the PreReqCall response. for example, if it is a JSON response, then set the JSON path to the required data in response.
- `void SaaSSvcReqParamKey::SetParameterType()`

  Sets that the data will be appended into actual endpoint as query parameter.
- `void SaaSSvcReqParamKey::SetReferencedType()`

  Sets that the data is referenced type, and hence will be cached by SDK and whenever the data is referred in actual endpoint, the cached data will be used.
- `void SaaSSvcReqParamKey::SetMaxValuesPerCall(simba_int16 value)`

  By default, the actual endpoint is formed for each data item found in PreReqCall response. This setting can be changed to allow multiple data items to be present in the actual endpoint as per your usage.

SaaSPreReqCall methods:

- `void SaaSPreReqCall::SetEndPoint(const Simba::Support::simba_string & in_endPoint)`

  Set the endpoint to be used to get the prerequisite data from server.
- `void SaaSPreReqCall::SetPaginationHandler (IPaginationHandler* in_handler)`

  If pagination is required to for PreReqCall endpoint, then set the pagination handler using this method.
- `void SaaSPreReqCall::SetParameterFormat(ParameterFormat value)`

  Set the parameter format. Possible values are:

  `PARAM_FORMAT_URL`: The parameter is embedded into URL.

  `PARAM_FORMAT_BODY`: The parameter is a part of API request body.

`PARAM_FORMAT_QUERY`: The parameter is a URI parameter (Default).

- `void SaaSPreReqCall::SetParserHandler(IParserHandler* in_handler)`

  Set the parser handler as per the Response data type for example., JSON, CSV, XML or Custom for PreReqCall.

- `void SaaSPreReqCall::SetHttpReqModifierHandler (IHttpRequestModiferHandler* in_handler)`

  If the HTTP request needs to be modified before it is sent to the server, then you should have implemented a request modifier handler by extending the `IHTTPRequestModifierHandler` interface. Register it for the current PreReqCall if required using this method.

- `void SaaSPreReqCall::SetHttpRespModifierHandler (IHttpResponseModiferHandler* in_handler)`

  If the HTTP response from server needs to be modified before it gets parsed by ParserHandler and get data, then you should have implemented a response modifier handler by extending the `IHTTPResponseModifierHandler` interface. Register it for the current PreReqCall if required using this method.

- `void SaaSPreReqCall::AddSvcReqParamKey(const SaaSSvcReqParamKey& in_reqParamKey)`

  Add the `SaaSSvcReqParamKey` for current PreReqCall.

- `void SaaSPreReqCall::AddPreReqCallPassdownColumn(const SaaSPreReqCallPassdownColumn& in_preReqCallPassdownColumn)`

  It specifies the passdownable column at the current level of prereqcall. For more details check *SaaSPreReqCallPassdownColumn* in *Simba REST SDK C++ API reference*.

- `void SaaSPreReqCall::SetListRoot(const Simba::Support::simba_wstring& in_listRoot)`

  Sets the list root for current PreReqCall.

- `void SaaSPreReqCall::SetPaginationData (SharedPtr<ISaaSPaginationData> in_paginationData)`

  Sets the pagination data for the current PreReqCall.

- `void SaaSPreReqCall::SetPreReqCall (AutoPtr<SaaSPreReqCall>& value)`

  PreReqCall can be nested with another PreReqCall which gets executed first before the outer PreReqCall.

What if prerequisite endpoint also needs some data to be feed into it before it is executed? This kind of scenario or usage is possible where one API endpoint is dependent on other API endpoint to get the prerequisite data and that prerequisite endpoint is again dependent on another prerequisite endpoint (defined under it's own scope) to get the prerequisite data. Such nested PreReqCalls can be set at the correct level of PreReqCall using this method.

- `void SaaSPreReqCall::SetSvcReqParamKey(const Simba::Support::simba_string& value)`

  Set the query parameter key to be appended into current PreReqCall request.

  PreReqCall example:

```
{
   SaaSReadApiEndpoint table_readApiEndpoint;
   table_readApiEndpoint.SetItemEndPoint("/rest/v1/items/
{{item_id}}");
  table_readApiEndpoint.SetListEndPoint("/rest/v1/items/
{{item_id}}");
  table_readApiEndpoint.SetType("PREREQ_CALLS");
  // Read API pre-reqcall
  {
     SaaSPreReqCall table_preReqCall;
     table_preReqCall.SetEndPoint
("/rest/v1/orders?sinceDate=TODAY");
     // ServiceReq param key read details
     {
       SaaSSvcReqParamKey table_svcReqParamKey;
       table_svcReqParamKey.SetKeyName("item_id");
       table_svcReqParamKey.SetSvcRespAttrField
("items.id");
       table_preReqCall.AddSvcReqParamKey(table_
svcReqParamKey);
```

```
      }

      table_preReqCall.SetListRoot("orderDetails");

      table_readApiEndpoint.SetPreReqCall(table_
preReqCall);

   }

   table_readApi.SetEndPoint(table_readApiEndpoint);

}
```

Skeleton Tables & Skeleton Columns

To get table and column metadata over REST API call, Skeleton Tables and Columns are used.

Skeleton Table and Columns can be defined using the same way the Static Tables and Columns are defined using classes `SaaSTable` and `SaaSTableColumn`, except that here we need to specify the endpoints to fetch the table and column metadata as well using `SaaSListVariable` class. The following methods of `SaaSListVariable` class can be used to specify endpoint information to get table or column metadata.

Methods of `SaaSListVariable` class:

- `void SetEndpoint(const Simba::Support::simba_string &value)` Set the endpoint used to fetch table or column metadata information.
- `void SetSvcRespAttrDefaultValue(Simba::Support::simba_string value)` Set the default value when corresponding field in response is empty.
- `void SetAcceptType(const Simba::Support::simba_string &value)` The accept type header to be passed in skeleton table or column request.
- `void AddVariables(const SaaSVariable &value)` Add variable into list of variables to be filled with their respective paths in http response.
- `void SetVariableRoot(const Simba::Support::simba_string &value)` Sets the root element of all variable paths in http response.

To specify which all details need to be fetched from the metadata endpoint API response, `SaaSVariable` class is used to specify the element path in API response. The following are the methods of `SaaSVariable` class:

- `void SetVariableName(const Simba::Support::simba_string &value)` Sets the variable name.
- `void SetSvcRespAttrMapping(const Simba::Support::simba_string &value)` Path to variable value in response.

Once defined, variables can be added using

`SaaSListVariable::AddVariables(const SaaSVariable &value)` method.

These variable names can be used in the places where we need to set data from API response such as column metadata (SQLType, Length, Name), table metadata (TableName). The variable names are ideally kept in placeholders {{ }} since these are used to refer to some particular data in the endpoint API response and these names also used in the table or column definition.

For Skeleton Tables, `SaaSListVariable` object can be added using the following method of `SaaSSkeletonTable`:

```
void AddListVariablesPrecalls(const SaaSListVariable& value)
```

For Skeleton Columns, `SaaSListVariable` object can be added using the following method of `SaaSSkeletonColumn`:

```
void SetListVariableAccess(const SaaSListVariable &value)
```

This way, by defining the variables and the endpoint infoormation, we can fetch the metadata information of table or column and store it in the variables. Now these variables can be used in table or column definition scope as well as at Read API access configuration level.

In Skeleton Column, `SaaSColumnDefinition` is used to specify the column metadata information (SQLType, Length, Precision, Scale, IsUnsigned) using the setter methods of `SaaSColumnDefinition`. Once defined, this can be added into Skeleton column using: the following method of `SaaSTableColumn`:

```
void SetColumnDefinition(const SaaSColumnDefinition &value)
```

Once the column is defined using `SaaSTableColumn`, it can be added into Skeleton Column using the following method of `SaaSSkeletonColumn`:

```
void SetSkeletonColumnDefinition(const SaaSTableColumn &value)
```

Skeleton column can be added into the Skeleton Table using following method of `SaaSTable`:

```
void AddSkeletonColumn(const SaaSSkeletonColumn& value)
```

Table definition once defined using `SaaSTable` can be added into the SkeletonTable using the following method of `SaaSSkeletonTable`:

```
void SetTableDefinition(const SaaSTable &value)
```

Example (Skeleton Table & Columns):

```
// Variable definition for Skeleton Column.
SaaSVariable skeleton_column_access_variable_1;
```

```
skeleton_column_access_variable_1.SetVariableName("
{{Length}}");
skeleton_column_access_variable_1.SetSvcRespAttrMapping
("//element/@max_length");


SaaSVariable skeleton_column_access_variable_2;
skeleton_column_access_variable_2.SetVariableName("
{{RestName}}");
skeleton_column_access_variable_2.SetSvcRespAttrMapping
("//element/@name")

 SaaSVariable skeleton_column_access_variable_3;
skeleton_column_access_variable_3.SetVariableName("
{{Type}}");
skeleton_column_access_variable_3.SetSvcRespAttrMapping
("//element/@internal_type");


// List Variable definition for Skeleton Column.
SaaSListVariable skeleton_column_variable_access;
skeleton_column_variable_access.SetEndpoint("
{{TableValue}}.do?SCHEMA");
skeleton_column_variable_access.SetAcceptType
("application/xml");
skeleton_column_variable_access.AddVariables(skeleton_column_
access_variable_1);
skeleton_column_variable_access.AddVariables(skeleton_column_
access_variable_2); skeleton_column_variable_
access.AddVariables(skeleton_column_access_variable_3);

// Column definition
SaaSTableColumn skeleton_column_1;
SaaSColumnDefinition skeleton_column_1_columnDefinition;
skeleton_column_1_columnDefinition.SetIsUnsigned("false");
skeleton_column_1_columnDefinition.SetLength("{{Length}}");
skeleton_column_1_columnDefinition.SetPrecision("38");
skeleton_column_1_columnDefinition.SetScale("3");
skeleton_column_1_columnDefinition.SetSQLType("{{Type}}");
skeleton_column_1.SetColumnDefinition(skeleton_column_1_
columnDefinition);
```

```
skeleton_column_1.SetName("{{RestName}}_value");
skeleton_column_1.SetSvcReqParamQueryMapping("{{RestName}}");
skeleton_column_1.SetSvcRespAttrListResult("{{RestName}}");
skeleton_column_1.SetSvcRespAttrItemResult("result.
{{RestName}}");
skeleton_column_1.SetPassdownable(true);

// Skeleton column definition (add SaaSTableColumn object and
the SaaSListVariable object)
SaaSSkeletonColumn skeleton_columns;
skeleton_columns.SetSkeletonColumnDefinition(skeleton_column_
1);
skeleton_columns.SetListVariableAccess(skeleton_column_
variable_access);

// Variable definition for Skeleton Table.
SaaSVariable table1_listvariable_1;
table1_listvariable_1.SetVariableName("{{TableName}}");
table1_listvariable_1.SetSvcRespAttrMapping("label");

SaaSVariable table1_listvariable_2;
table1_listvariable_2.SetVariableName("{{TableValue}}");
table1_listvariable_2.SetSvcRespAttrMapping("value");

// List Variable Definition for Skeleton Table.
SaaSListVariable table1_listvariable; table1_
listvariable.AddVariables(table1_listvariable_1);
table1_listvariable.AddVariables(table1_listvariable_2);
table1_listvariable.SetAcceptType("application/json");
table1_listvariable.SetEndpoint("/api/now/doc/table/schema");
table1_listvariable.SetVariableRoot("result");

// Table Definition (using SaaSTable object)
SaaSTable table1;
table1.AddSkeletonColumn(skeleton_columns);
table1.SetTableName("{{TableValue}}");
table1.SetTableSchemaName("Actual");
table1.AddItemEndpointColumnNames("sys_id_value");
table1.AddItemEndpointColumnNames("sys_id_display_value");
table1.SetSortable();
table1.SetPageable();
```

```
table1.SetValPath("value");

// SaaSTable Read API access configuration will follow same
as it
// is done for normal (Non Skeleton) Table.

// Skeleton Table definition (Add SaaSListVariable object and
SaaSTable object)
SaaSSkeletonTable skeleton_table1;
skeleton_table1.AddListVariablesPrecalls(table1_
listvariable);
skeleton_table1.SetTableDefinition(table1);

// Add Skeleton Table in SaaSConfiguration object.
// Set Skeleton Table to Not initialized since this will be
initialized at run time.
out_configuration.SetSkeletonTableInitialized(false);
out_configuration.AddSkeletonTable(skeleton_table1);
```

For more details, please check `SaaSSkeletonTable`, `SaaSSkeletonColumn`, `SaaSColumnDefinition`, `SaaSListVariable`, `SaaSVariable` in C++ API reference.

Custom Extensions

Some Essential features to work with REST API are Authentication, Pagination, Parsers. SDK supports all these three features by default. But these are limited to few Types namely:

Authentication:

- Basic
- OAuth 2.0
- Access Token

Pagination:

- Index Based – EMPTY PAGE, OFFSET BASED
- Response Based – HEADER, BODY, TOKEN

Parsers:

- JSON
- XML
- CSV

So, if the requirement is to support any other type apart from the above mentioned types, then you need to implement your own Handler for that type by extending the Base Class.

In this section, we will go through the syntax of each Custom Extensions (`Authentication, Pagination, Parser, Request/Response Modifier`), and in addition to that any other classes that need to be implemented with one example of each.

Authentication

If your custom connector needs to support Authentication type that is not supported by SDK by default, then you need to implement authentication by extending the `IAuthenticationHandler` and `IAuthenticationContext` classes.

An Authentication Handler is shared among multiple connections. So, it is important to track that the calls to the methods of authentication handler came from which particular connection session, for example, context needs to be identified. This can be managed by keeping a authentication context for each connection and for each authentication context which is the associated connection object. And then any

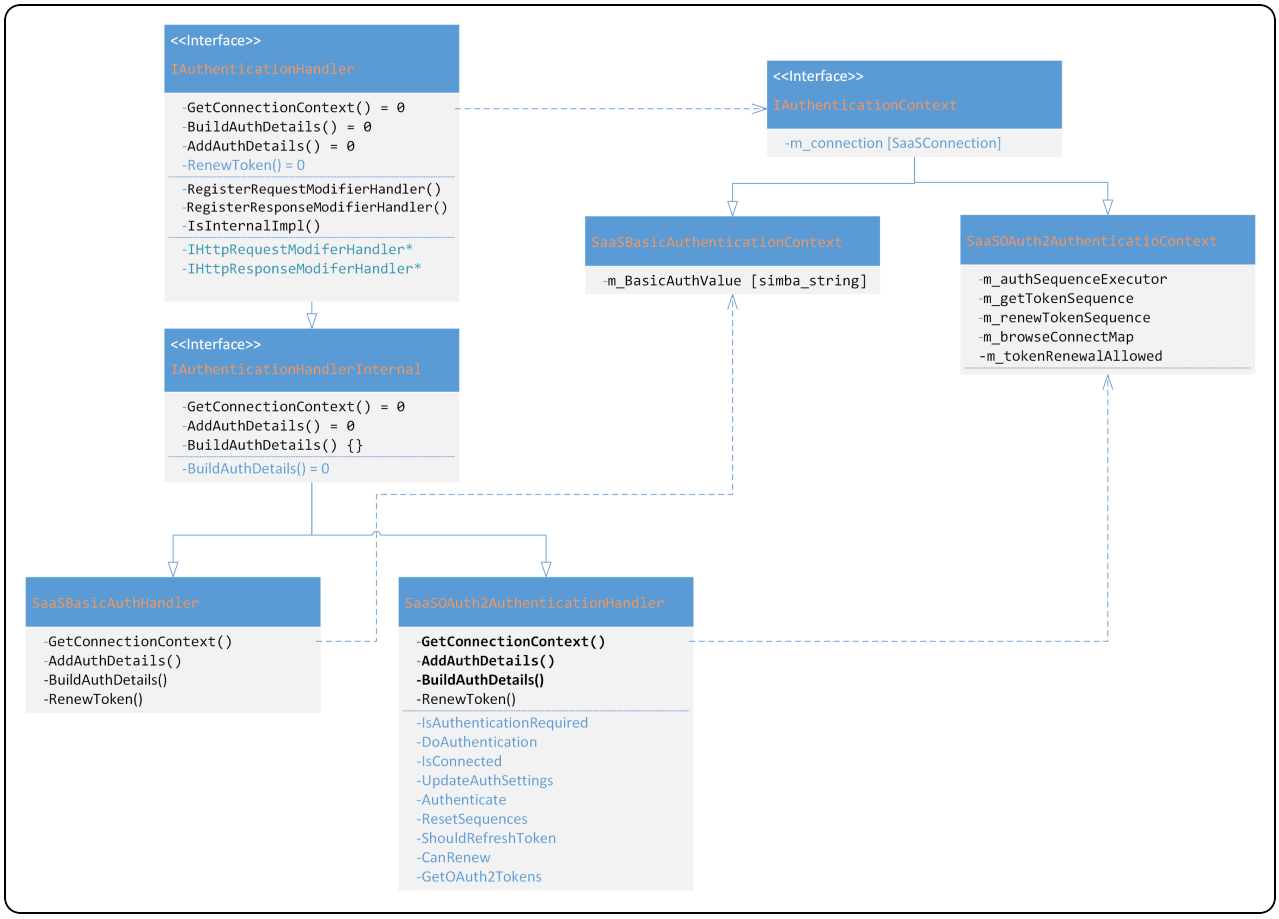
requests for authentication are served based on the authentication context passed to the Authentication Handler.

When we attempt to connect:

Connection object calls `GetConnectionContext`() method of AuthenticationHandler to get the authentication context.

Pointer to connection object is stored in the authentication context.

For the `BuildAuthDetails/AddAuthDetails/RenewToken` caller would pass `IAuthenticationContext` object, which will contain the connection object that holds the auth connection context.



Authentication context is created for each connection session to store the authentication details generated for a chosen authentication type which can be used across multiple methods of `IAuthenticationHandler` discussed below since context is passed to each of them as a parameter.

To create an authentication context for custom authentication type:

1. Extend the `IAuthenticationContext` interface.
2. Declare variables which may be used across multiple methods of your authentication handler (specifically `BuildAuthDetails` method is used to build the authentication details and store it under authentication context and later `AddAuthDetails` method reads these authentication details to be added into the HTTP request).

To create a Custom Authentication Handler:

1. Extend the `IAuthenticationHandler` interface.
2. Implement the below methods of the interface in your Custom Authentication Handler:

   - `virtual AutoPtr<IAuthenticationContext> GetConnectionContext() = 0;`
   - Create and return a new instance of authentication context for the authentication type that this authentication handler is created for.

```
virtual void BuildAuthDetails(
                const Simba::DSI::DSIConnSettingRequestMap&,
                IAuthenticationContext&,
                Simba::DSI::DSIConnSettingResponseMap&) = 0;
```

   Builds and stores the required authentication details in the connection context for fetching data from Data-Source.

   - `virtual void AddAuthDetails(`
   - `SaaSRequestData&,`
   - `IAuthenticationContext&) = 0;`
   - Adds details to the Request object, built using the `BuildAuthDetails` call.
   - `BuildAuthDetails` should always be called first which would have stored required values for `AddAuthDetails` to work, in connection context.
   - `virtual void RenewToken(IAuthenticationContext&) = 0;`
   - RenewToken will be called when SDK comes across errors pointing token renewal need, or in case of auto-renewal renewal timing is almost reaching.

Now we will create a Custom Authentication Handler step by step:

1. **Create Authentication Context Class**: Extend the IAuthenticationContext interface and create Custom Authentication Context for the Custom Auth type.

```
class CustomAuthenticationContext : public
IAuthenticatiionContext { public: // Declare variables
```

```
which will store the // Authentication details to be used
later during // AddAuthDetails method call. simba_string
token;  }
```

Note: Also declare any variables which may be used to get authentication details, for example, in OAuth 2 context, we kept `BrowseConnectMap`, `SaaSAuthSequenceExecutor`, `GetToken` and `RenewToken` sequence since those details will be used to get the new Access token. So, if any variable needs to be declared specifically for authentication type, then declare it in the authentication context.

2. **Create Authentication Handler**:Extend the interface `IAuthenticationHandler` to create custom authentication handler and implement the full functionality under the correct methods of `IAuthenticationHandler`.

   CustomAuthenticationHandler.h (Header file):

```
/// Custom Authentication Context

class CustomAuthenticationContext : public
IAuthenticatiionContext

{

public:

// Declare variables which will store the

// Authentication details to be used later during

// AddAuthDetails method call.

simba_string token;

}

/// Custom Authentication Handler

class CustomAuthenticationHandler : public
IAuthenticationHandler

{

// Default Constructor.

CustomAuthenticationHandler();

// Declare the 4 pure virtual Methods of
```

```
// IAuthenticationHandler here as mentioned above as
// well as any Custom methods if needed.
}
```

CustomAuthenticationHandler.cpp (Definition file):

```
#include "CustomAuthenticationHandler.h"
 // Any additional include files as required.
```

Implement the following methods in CustomAuthenticationHandler.cpp.

GetConnectionContext():

Create and return a new instance of customauthenticationcontext (the class that we created earlier in CustomAuthenticationHandler.h header file). A lock need to be acquired since authentication handler is shared across multiple connections, but each connection has a separate authentication context.

```
AutoPtr<IAuthenticationContext>
CustomAuthenticationHandler::GetConnectionContext()
{
/// Taking lock before creating Connection Auth Context,
/// as the handler is shared by multiple connections
CriticalSectionLock lock(m_criticalSection);
return AutoPtr<IAuthenticationContext>(new
CustomAuthenticationContext());
}
```

BuildAuthDetails

Inputs to this method are:

- **Connection String parameters and their values** (`in_connectionSettings`)

  Used to verify the input connection string to see if all required or optional parameters are provided or not. If required parameters are missing, use `DSIConnection::VerifyRequiredSetting` or use `DSIConnection::VerifyOptionalSetting` method to verify whether

optional parameter is present or not. If any of the required or optional setting verified using above methods are not present, then it gets automatically added to the `out_connectionSettings`

- **Authentication Context** (`out_connectionContext`)

  Used to store the authentication details for current authentication context.

- **Connection settings object** (`out_connectionSettings`) to store the connection parameters which are missing from the connection string.

Assuming that the Access Token needs to be provided in the connection string for our custom authentication to work, we can check in this method whether access token is provided or not, if provided we will copy the value and store it in out_connectioncontext and if not provided, we will use "`VerifyRequiredSetting`" method to store it in the `out_connectionSettings` variable.

```
void BuildAuthDetails(
const Simba::DSI::DSIConnSettingRequestMap&
in_connectionSettings,
IAuthenticationContext& out_connectionContext,
Simba::DSI::DSIConnSettingResponseMap&
out_connectionSettings)
{
CustomAuthenticationContext& authContext =
static_cast<CustomAuthenticationContext&>(out_
connectionContext);
DSIConnSettingRequestMap::const_iterator
connectionSettingItr;
// Check if Auth_AccessToken is provided.
connectionSettingItr =
in_connectionSettings.find(CONN_AUTH_ACCESS_TOKEN_KEY);
if (in_connectionSettings.end() !=
connectionSettingItr && !connectionSettingItr->second.
.GetStringValue().empty())
```

```
{
authContext.accessToken = connectionSettingItr->second.
.GetStringValue();
return;
}
// Check if EncAccessToken is provided then decrypt its value.
connectionSettingItr =
in_connectionSettings.find(SAAS_ENC_ACCESSTOKEN_KEY);
if (in_connectionSettings.end() !=
connectionSettingItr && !connectionSettingItr->second.
.GetStringValue().empty())
{
DriverSupport::DSEncryptionUtils::DecryptFromHex(
connectionSettingItr->second.GetStringValue(),
authContext.accessToken,
SAAS_ENC_PROPS);
}
else
{
// None of the Auth_AccessToken or EncAccessToken are
// provided. So add it in out_connectionSettings.
DSIConnection::VerifyRequiredSetting(
CONN_AUTH_ACCESS_TOKEN_KEY,
in_connectionSettings,
out_connectionSettings,
false);
```

```
return;
}
}
```

AddAuthDetails

Inputs to this method are:

- `SaaSRequestData` which is used to set the properties of HTTP request (i.e., URL, path, Query parameters etc).
- `IAuthenticationContext` to read the Authentication details to be set in the HTTP request.

In our case, we stored the access Token value in Authentication context during `BuildAuthDetails` call and here we will add it at proper place in HTTP request (i.e. Request Headers) (Assuming token Type is `Bearer`)

```
void SaaSCustomAuthenticationHandler::AddAuthDetails(
SaaSRequestData& io_request,
IAuthenticationContext& in_connectionContext)
{
SaaSCustomAuthenticationContext& authContext =
static_cast<SaaSCustomAuthenticationContext&>
(in_connectionContext);
Headers authHeader;
std::stringstream headerValue;
headerValue << "Bearer " << authContext.accessToken;
authHeader.insert(make_pair(simba_string("Authorization"),
headerValue.str()));
io_request.SetHeaders(authHeader);
}
```

RenewToken

Called when server throws an error related to token expire and token needs to be renewed to continue sending more requests to server.

Inputs to this method are:

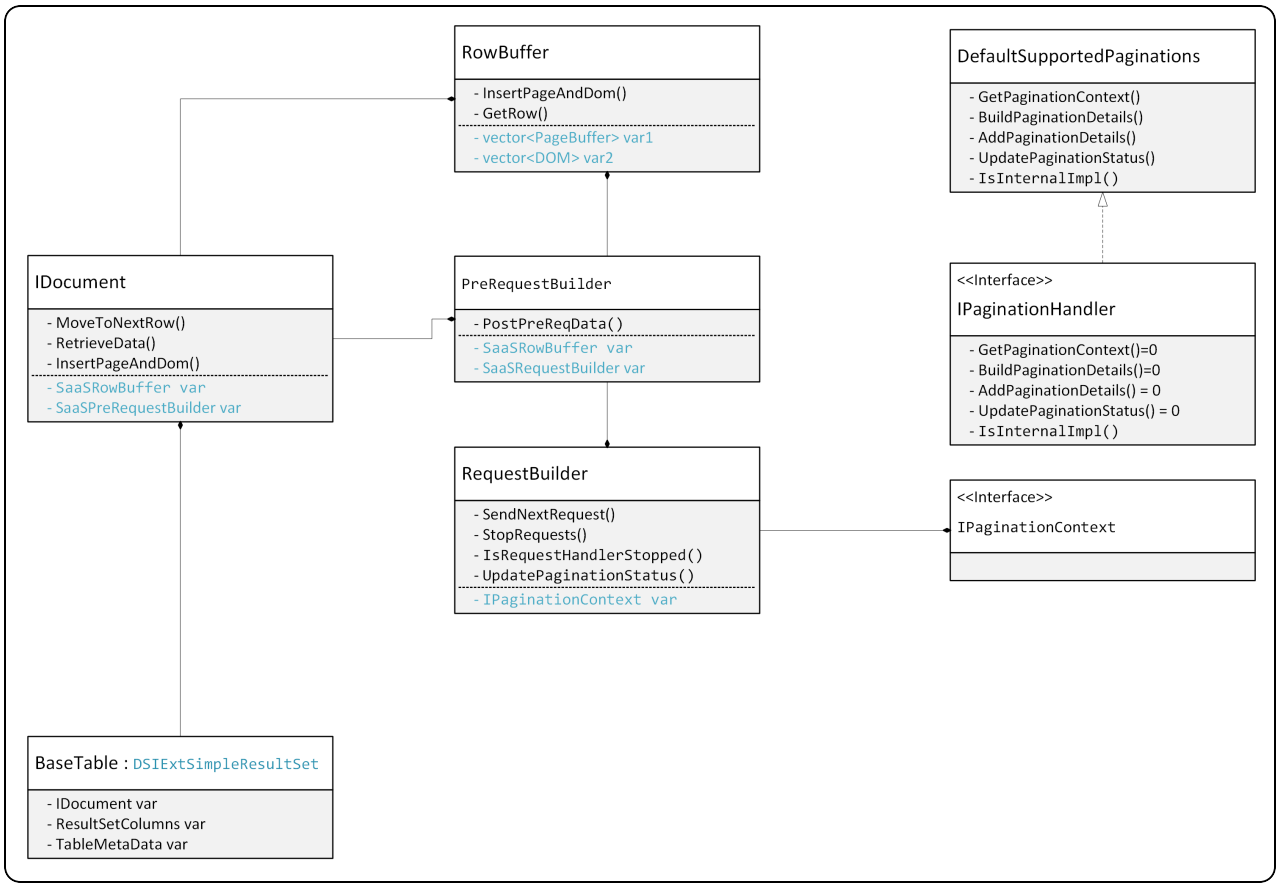
1. `IAuthenticationContext` to read the renew token details.

```
void SaaSCustomAuthenticationHandler::RenewToken(
IAuthenticationContext & in_connectionContext)
{
/// Do nothing
}
```

If Renew Token flow is not supported from datasource end, then no need to implement this method (i.e., just keep empty body).

Pagination

If custom connector needs to support pagination type that is not supported by SDK by default, then custom pagination handler needs to be implemented.



Each pagination type needs its own specific details to be provided to make it work. Besides that, there is common information which can be used across all pagination types. For example, pagination status is generalized across all pagination types to signal the status of pagination at any moment.

SaaSPaginationStatus (Type: enum)

`PAGING_NEXT_INFO_AVAILABLE` (Default)

If the next page information is available, and next request can be built. This means that you have captured the next page information and are ready to send the next page of request to server.

`PAGING_NEXT_INFO_AWAITED`

Next page information is awaited, usually for the case where next page information is present in the arriving page. Usually this will be set once you have formed the current page information but next page information is not yet available.

`PAGING_BLOCK_SENT`

Some pagination information can be calculated without next page information. This can lead to requests being sent continuously, irrespective of the fact that client consumption of data possibly be slower. So once the block of requests is sent (to be determined by the block size), the pagination status can be changed to this to check if we are done or need to send more requests.

`PAGING_COMPLETE`

Pagination complete, no more pages required to be sent after this one.

`PAGING_SUCCESS`

Pagination success, maintained for representing the success cases.

`PAGING_SUCCESS_EMPTY_PAGE`

Pagination success but received empty page.

`PAGING_EXCEPTION`

Pagination exception, maintained for any exceptions handling.

`PAGING_ERROR`

Pagination handler encountered error.

TerminationType (Type: enum)

Termination type determines how the pagination will terminate once started. There are few possible ways for index based and response based pagination supported by SDK:

`TYPE_ROWCOUNT`

Total number of rows available in the response of the API. Once the total number of rows fetched is equal to the total number of rows available from API, the pagination is terminated.

`TYPE_PAGECOUNT`

Total number of pages, for the row count sent. Once the total number of pages sent is equal to the total number of pages that needed to be sent, pagination is terminated.

`TYPE_EMPTYPAGE` (Default)

If the last page number is not available via API, then API automatically sends the empty page response, once all the available data is fetched using API. So empty page indicates end of pagination.

`TYPE_NEXTPAGEURL`

Next page URL fetched from the API response. Pagination terminates when it is empty or not present in API response.

`TYPE_TOKEN`

Token to be used to form the next page URL and is taken from the API response. Pagination terminates when it is empty or not present in API response.

OffsetType (Type: enum)

When using index based pagination, the OffsetType needs to be set to either of the following values.

`OFFSET_ITEM`

Each item or the row is considered while calculating the page size.

`OFFSET_PAGE`

Group of items represented by the offset element, page with limit.

`OFFSET_NONE` (Default)

No possible offset type.

TerminationKeyPresentIn:

Used to specify where the pagination termination key is present when using response based pagination or index based pagination.

`KEY_HEADER`

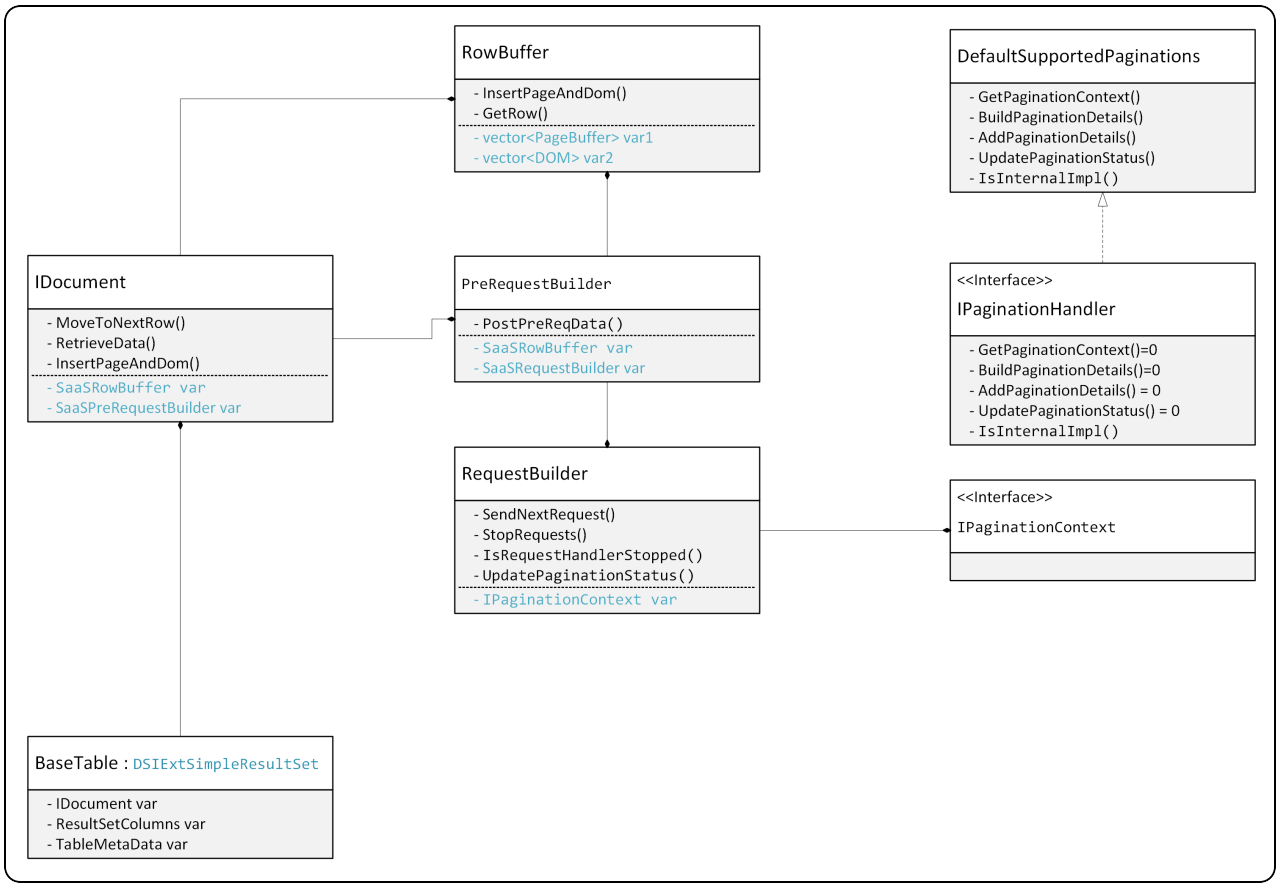
The key value is present in the http headers of response.

`KEY_BODY`

The key values are present in the http response body.

`KEY_NONE` (Default)

Key value is not available, can be used for `EMPTYPAGE`.



Pagination Handler has a context (`IPaginationContext`) where it stores the current connection object reference (`SaaSConnection`) and stores the variables whose values may be determined later during pagination or values which are required to be set for each page request (eg. LastPage, PageSize, Offset). These details are calculated by the pagination handler based on the Input pagination parameters given to it.

Pagination Context stores the `SaaSPaginationStatus` and the `ISaaSPaginationData. ISaaSPaginationData` interface is used to create a pagination data class wherein we keep all the Input pagination details and getters or setters for it for a particular pagination type to accept the pagination data input from the Connector Configuration.

Data Source specific pagination details are kept under pagination data of a specific pagination type and the pagination details that are calculated by pagination handler based on the input pagination data are kept under the pagination context of a specific pagination type.

Let us walk through the code to understand how to implement a custom pagination handler for a pagination type not supported by SDK:

1. Create a class to store pagination data for your custom page handler by extending the ISaaSPaginationData interface. This class will have all the pagination input variables declared with getters and setters required to set the datasource specific pagination data via Connector Configuration and get or read the data in Pagination Handler to handle pagination.

   For exampl, If a custom pagination type requires three inputs, TerminationType, a query parameter key for some pagination attribute, and it's initial value.

   File: `SaaSCustomPaginationData.h`

```
#include "Pagination/ISaaSPaginationData.h"

class SaaSCustomPaginationData : public
ISaaSPaginationData
{
public:
SaaSCustomPaginationData () :
m_queryParm(“”),
m_initialValue(“”),
m_terminationType(TYPE_EMPTYPAGE)
{
// Do Nothing.
}
virtual ~SaaSCustomPaginationData()
{
// Do Nothing.
}
inline const simba_string& GetReqQueryOffsetKey()
{
return m_queryOffsetKey;
}
inline void SetReqQueryOffsetKey(const
simba_string&
in_queryOffsetKey)
{
m_queryOffsetKey = in_queryOffsetKey;
}
inline const TerminationType&
GetTerminationType()
```

```
{
return m_terminationType;
}
inline void SetTerminationType (const
TerminationType&
in_terminationType)
{
m_terminationType = in_terminationType;
}
inline const simba_string& GetQueryLimitKey()
{
return m_queryLimitKey;
}
inline void SetQueryLimitKey (const
simba_string& in_queryLimitKey)
{
m_queryLimitKey = in_queryLimitKey; }
private:
simba_string m_queryOffsetKey;
simba_string m_queryLimitKey;
TerminationType m_terminationType;
};
```

2. Create pagination context for custom pagination type. Extend the `IPaginationContext` interface and add any variables which will be used internally by custom pagination handler to calculate pagination details, set it and then read later when required.

   `IPaginationContext` has the following properties:

   - Connection object as reference (`SaaSConnection`). (Default: `NULL`)
   - `ISaaSPaginationData` refers to a pagination data of a pagination type.
   - Last page number sent to server (initial page starts with index 1).
   - Boolean variable indicating whether pagination terminated or not (Default: `false`)
   - `SaaSPaginationStatus` to know the pagination state. (Default: `PAGING_NEXT_INFO_AVAILABLE`)
   - `IPaginationContext` has following methods available:
     - `void SetConnectionReference(SaaSConnection*)` to set the connection reference.

- `SaaSConnection* GetConnectionReference()` to get the connection reference.

`SaaSCustomPaginationContext` may require some variables to be declared and initialized in pagination context , for example,LastPage, PageSize, Offset.

```
#include "SaaSSQL.h"

#include "Pagination/IPaginationContext.h"

class SaaSCustomPaginationContext : public
IPaginationContext

 {

public:

 /// The limit value for the query.

simba_uint64 m_limit;

/// The offset value query.

 simba_uint64 m_offset;

 public:

 SaaSCustomPaginationContext() :

m_limit(0),

 m_offset(0) {}

};
```

3. Create pagination handler for the custom pagination type. Extend the `IPaginationHandler` interface and implement the below abstract methods:

   `virtual AutoPtr<IPaginationContext> GetPaginationContext() = 0;`

   Creates and returns a URL or table related pagination context. Implementation of this method should take care of multi threaded case returning new context whenever called. Connections to make sure, having only one connection context. Deletion of connection context will be taken care of by the caller.

   - `virtual void BuildPaginationInfo( ISaaSPaginationData& in_paginationinfo,IPaginationContext& in_ paginationContext, simba_uint64 in_pageSize) = 0;`

Pagination handler to prepare for the pagination information to be added later on during `AddPaginationDetails` call.

- `virtual SaaSPaginationStatus AddPaginationDetails( SaaSRequestData& io_request, IPaginationContext& in_paginationContext) = 0;`

  Adds next page - pagination detail to the http request data. `io_request` is updated with added pagination detail for next page. This method returns status of the possibility of adding next page information.
- `virtual SaaSPaginationStatus UpdatePaginationStatus( SaaSPaginationUpdate& in_status, IPaginationContext& in_paginationContext) = 0;`

  Update pagination based on response of previous request, update request and send.

  `in_status` is the status of the pagination request previously sent. Returns status of the process pagination status, some cases like empty page might not be an error for the pagination type.
- `virtual SaaSPaginationStatus GetPaginationState (IPaginationContext& in_paginationContext) = 0;`

  Returns the pagination handler state. Will be used by RequestBuilders to decide if more pages can be sent for AWAITED and BLOCK_SENT pagination statuses.
- `virtual simba_uint64 GetBlockSize(IPaginationContext& in_paginationContext) = 0;`

  Get the block size if any for the pagination handler. As for some of the pagination types it's possible to prepare requests to be sent till the possible last page. But that would mean extra memory usage, as not all pages are required to be fetched upfront. So, the block size helps in keeping the memory usage in check.

SaaSCustomPaginationHandler.h

```
#include "SaaSSQL.h"
#include "Pagination/IPaginationHandler.h"
#include "SaaSCustomPaginationContext.h"
 #include "CriticalSection.h"
```

```
 namespace Simba
 {
 namespace SIMBARESTSDK
 {
class SaaSCustomPaginationData;
 class SaaSRequestData;
 }
}
 namespace Simba
 {
 namespace SIMBARESTSDK
 {
class SaaSCustomPaginationHandler : public IPaginationHandler
 {
 public:
 /// @brief Constructor.
///
 /// @param in_connection SaaSConnection instance.
SaaSCustomPaginationHandler();
/// @brief Destructor.
 ~SaaSCustomPaginationHandler() {};
 public:
 virtual AutoPtr<IPaginationContext> GetPaginationContext();
 virtual void BuildPaginationInfo( ISaaSPaginationData& in_
paginationinfo, IPaginationContext& in_paginationContext,
simba_uint64 in_pageSize);
```

```
 virtual SaaSPaginationStatus AddPaginationDetails(
SaaSRequestData& io_request, IPaginationContext& in_
paginationContext);

 virtual SaaSPaginationStatus UpdatePaginationStatus(
SaaSPaginationUpdate& in_status, IPaginationContext& in_
paginationContext);

 virtual SaaSPaginationStatus GetPaginationState(
IPaginationContext& in_paginationContext);

 virtual simba_uint64 GetBlockSize(IPaginationContext& in_
paginationContext);

 private:

 /// Critical section used for concurrency creating

 /// Pagination Context.

 CriticalSection m_criticalSection;

 };

 }

 }
```

SaaSCustomPaginationHandler.cpp

```
/// Constructor

 SaaSCustomPaginationHandler:: SaaSCustomPaginationHandler ()

 {

 }

 /// GetPaginationContext()

 AutoPtr<IPaginationContext>
SaaSCustomPaginationHandler::GetPaginationContext()

 {

 CriticalSectionLock lock(m_criticalSection);
AutoPtr<IPaginationContext> customPaginationContext(new
SaaSCustomPaginationContext());

 return customPaginationContext;
```

```
 }

 /// BuildPaginationInfo()

 void SaaSCustomPaginationHandler::BuildPaginationInfo(
ISaaSPaginationData& in_paginationinfo,

 IPaginationContext& in_paginationContext,

 simba_uint64 in_pageSize)

 {

 SaaSCustomPaginationContext& paginationContext = dynamic_
cast<SaaSCustomPaginationContext&>

 (in_paginationContext);

 paginationContext.m_paginationInfo =

 dynamic_cast<SaaSCustomPaginationData*>(&in_paginationinfo);

 // PreCalculated page size by SDK To support variable page
size

 // based on no. of columns to be fetched to speed up the
data

 // fetch.

 paginationContext.m_pageSize = in_pageSize;

}

 /// AddPaginationDetails()

 SaaSPaginationStatus
SaaSCustomPaginationHandler::AddPaginationDetails(

 SaaSRequestData& io_request,

IPaginationContext& in_paginationContext) {
SaaSCustomPaginationContext& paginationContext =

dynamic_cast<SaaSCustomPaginationContext&>

 (in_paginationContext);

 SaaSCustomPaginationData& paginationData =
```

```
 dynamic_cast<SaaSCustomPaginationData&>
(*(paginationContext.m_paginationInfo));
 /// Page size needs to be sent for all the page requests.
 {
 simba_uint64 limitQuery = paginationContext.m_pageSize;
 if (paginationContext.m_limit > 0)
 {
limitQuery = (paginationContext.m_limit >=
 (((paginationContext.m_currentPage - 1) *
 paginationContext.m_pageSize) +
 paginationContext.m_pageSize)) ?
 paginationContext.m_pageSize :
 (paginationContext.m_limit %
 paginationContext.m_pageSize);
 if (0 == limitQuery)
 {
 in_paginationContext.m_paginationStatus =
 PAGING_COMPLETE;
 return in_paginationContext.m_paginationStatus;
 }
 }
 simba_string pageQueryValue(std::to_string(limitQuery));
 io_request.SetQueryParameter(paginationData.
 GetQueryLimitKey(), pageQueryValue);
 }
 simba_string pageQueryValue =
```

```
 std::to_string(paginationContext.m_currentPage);

 io_request.SetQueryParameter(

 paginationData.GetReqQueryOffSetKey(), pageQueryValue);

 in_paginationContext.m_paginationStatus =

 PAGING_NEXT_INFO_AWAITED;

 }

 // Increase the page count for the next request to be sent
paginationContext.m_currentPage++;

 return in_paginationContext.m_paginationStatus;

 }
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
/// SaaSPaginationStatus
SaaSCustomPaginationHandler::UpdatePaginationStatus(
SaaSPaginationUpdate& in_status,

 IPaginationContext& in_paginationContext)

 {

 SaaSCustomPaginationContext&

paginationContext =

dynamic_cast<SaaSCustomPaginationContext&> (in_
paginationContext); SaaSCustomPaginationData& paginationData
=

dynamic_cast<SaaSCustomPaginationData&>(*

(paginationContext.m_paginationInfo));

 TerminationType terminationType =
paginationData.GetTerminationType();

 // If the response body is empty -

 // - If it's the first page, stop pagination.

 // - Else

 if (NULL == in_status.m_parsedDoc)
```

```
 {
 if (TYPE_EMPTYPAGE == terminationType)
{
 paginationContext.m_pagingComplete = true;
 paginationContext.m_paginationStatus =
PAGING_COMPLETE;
 return PAGING_SUCCESS;
 }
 }
 paginationContext.m_paginationStatus =
 (PAGING_COMPLETE == paginationContext.m_paginationStatus) ?
PAGING_COMPLETE : PAGING_NEXT_INFO_AVAILABLE;
 return PAGING_SUCCESS;
 }
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
/// SaaSPaginationStatus
SaaSCustomPaginationHandler::GetPaginationState
(IPaginationContext& in_paginationContext)
 {
return in_paginationContext.m_paginationStatus;
 }
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
/// simba_uint64
SaaSCustomPaginationHandler::GetBlockSize(IPaginationContext&
in_paginationContext)
 {
 return 10; // Fix me
 }
```

Parser

The REST API response needs to be parsed to get the intended data out of it. Based on the response type SDK supports by default parsing for three types of data namely JSON, XML and CSV. If you need to parse any other response type, then SDK provides the parser extension using which you can implement your own Parser for the response type.

Parsing includes the following steps:

1. Create a response container of specific response type. `IParsedDoc` provides an interface which you can extend to create a container or document to store the API response of intended type. Forexample, for JSON response, `rapidjson::Document` is created.
2. Parse the response to capture the original API response in the response container created in Step 1. `IParserHandler` provides an interface by which you can parse the API response and store it in the container of a specific response type.
3. Once the response is parsed, it can be used to get the rows of data. Get data for each data cell or column in a row by identifying the data path for it in the parsed response. Data path will be provided in the configuration for each column in a table. `SaaSDataCell` is used to capture the single cell data for current row. `Row` is used to capture a single row of data. `PageRows` is used to capture the rows of data got from the current parsed response.

In addition to above steps, an ExceptionHandler need to be implemented for the response type to parse the errors in API response so that the exact API errors will be thrown based on the path to the error messages and error codes in the response provided in configuration. `IExceptionHandler` provides an interface to implement this functionality. (Note: This needs to be implemented only for Custom Parsers for response types not supported by SDK).

Create a Document to store parsed response

A document or response container needs to be created first to store the parsed response (once it gets parsed in the later steps). `IParsedDoc` is the base interface to be used to create a document for any response type. You can set or get the content type of the response using this interface as well as get the reference to the parsed response document or container.

Extend the `IParsedDoc` interface and implement the abstract method `GetParsedDoc`() to return pointer to the newly created document to store the parsed response.

```
void* GetParsedDoc()
```

```
{
// Code to create a new Document of intended response type
// and return a pointer to newly created document.
}
```

Exception Handling for API errors

A custom exception handler need to be created to handle API level errors for the specific response type. Extend the `IExceptionHandler` interface and implement the below abstract methods:

- Extract the error message and error code from the error response using:

  ```
  virtual simba_wstring EvaluateErrorResponse(
  const IParsedDoc& in_parsedResponse,
  const std::vector<SaaSErrorPath>& in_errorPaths)
  const = 0;
  ```

This method finds the error messages and error codes in the response by paths given in input (`in_errorPaths`) and returns the error message.

```
virtual simba_wstring EvaluateErrorInPayload(
const IParsedDoc& in_parsedResponse,
const std::vector<SaaSErrorPath>& in_errorPaths,
ErrorMessages& out_warningMessageContainer) const = 0;
```

This method is also used to extract the error message from response same as `EvaluateErrorResponse`()method. But this method accepts one additional parameter `ErrorMessages` which is a vector of strings representing the errors which would not be thrown but will be handled by warning Listener.

- ```
  virtual bool EvaluateError(
  SaaSExceptionHandlingInfo& in_handlingInfo,
  const IParsedDoc& in_parsedResponse,
  const simba_string& in_respString,
  const simba_int32 in_httpCode,
  const std::vector<SaaSErrorPath>& in_errorPaths) const = 0;
  ```
  ```

Evaluate the parsed response, http response code, exception handling information and the path to errors in the response to tell whether the error occurred or not. `SaaSExceptionHandlingInfo` has the following info:

- `IWarningListener` to consume extra errors as warnings.
- Boolean var to indicate whether to throw an error got from API response. (Default: false)
- Boolean var to indicate whether the error is in the response payload. (Default: false)
- Error message in string format.

`in_parsedResponse` is the parsed response to look for API errors.

`In_respString` is the API response string used to throw an error in case the complete response is to be shown as error message.

`in_httpCode` is the http response code. This code can be used to check if API error occurred or not by comparing it with the known API error codes.

`in_errorPaths` is the list of paths to look for API errors in the parsed response. For more details check SaaSErrorPaths in the C++ API reference.

Return true if the error occurred, false otherwise.

Parser Handler

`IParserHandler` Interface is for defining required functionality to be implemented for parser handling for the Data-Source. Apart from SDK provided main-stream parser types like JSON, XML etc. Users might want to extend this interface, to handle any proprietary response type.

It covers the functionality to get the intended data out of the parsed API response given the path to it in the API response. Let us walkthrough the abstract methods that need to be implemented of this interface to implement the custom parser handler.

IParserHandler(IExceptionHandler& in_exceptionHandler)

The constructor of IParserHandler accepts exception handler as parameter. So, create an exception handler and pass it to IParserHandler in the constructor of your Custom Parser Handler class.

```
virtual AutoPtr<IParsedDoc> ParseResponse(

const SaaSResponseData& in_ responseData,

Simba::Support::simba_string& out_error) = 0;
```

Parse the API response string into a response container or document associated with a specific response type. There are thirdparty libraries available in the market to parse the type of data, for example, to parse JSON data SDK uses `RapidJSON` library, to parse XML it uses `PugiXML` library to identify the library to be used for your use case, if required.

The API response string can be taken using `GetBody`() method of `SaaSResponseData` for example, `in_responseData.GetBody()`.For more details on `SaaSResponseData` in C++ API reference.) and then it can be used to provide as input to one of the library function which accepts string as input to parse the API response and returns the response container or document representation of it in the same format as the response type.

For example, RapidJSON provides a parse function which takes UTF8 string as input and stores the parsed response in rapidjson document.

Return the pointer to parsed response container or document if parsing is successful.

Some libraries also provide a function to check for parsing errors if occurred any, if so it can be captured in the `out_error` variable and in that case return NULL from this method.

```
virtual PageRowsDataAutoPtr GetRowsFromDoc(

IParsedDoc& in_parsedDoc,

SaaSColumnParsingConstraints& in_colConstraints,

const simba_wstring in_listPath = "") = 0;
```

Get all the rows from the parsed document and return a pointer to newly created rows of data got from the current parsed document.

`SaaSColumnParsingConstraints` provides constraints to get column data like it provides the path to the column data in the API response (. (dot) separated or / (slash) separated path based on the type of data).

To get the list of column paths, use `GetColumnPaths`() method of `SaaSColumnParsingConstraints`. You can also check whether a column is a primary key or not by passing the column's path as parameter to the method: `SaaSColumnParsingConstraints::IsPrimaryKey(const simba_wstring& in_colName)`.

For more details on `SaaSColumnParsingConstraints` please check the C++ API reference.

The third parameter to this method `in_listPath` will be provided if the data is in. If rows are contained under a parent path. for example, for json below rows is a list path –

```
{
"rows" : [
{"row1-cell1", "row1-cell2"},
{"row2-cell1", "row2-cell2"},
{"row3-cell1", "row3-cell2"}
]
}
```

Note on Get Rows

A column path provided may be present in multiple places in the same API response, For instance, there could be information (i.e. name of an item) that exists multiple times in a single API request to get List of items. So, each occurrence of Item name corresponds to column data for one particular row. That means the number of occurrences of Item name in API response, those many rows of data can be returned.

As mentioned earlier, each `Row` corresponds to Vector of SaaSDataCell. And each `SaaSDataCell` corresponds to single data cell holding data of a particular column in a particular row.

Data is stored as a void pointer, so once you Iterate over your API response and reach to some data as per the column path provided, then store that data as void pointer in `Row` object at correct index. The index number of row object corresponds to the column index so it is very important to identify the column index same as column path index in column paths vector. So, if at the 0th index, the column path of itemname column is stored, then at the 0th index only the data of itemname column should be stored in the row.

Likewise, once you have got all the rows from the current parsed response, it can be stored in a `PageRows` and return auto pointer to that `PageRows` from this method. For more details on `SaaSDataCell`, `Row`, `PageRows` and `PageRowsDataAutoPtr` check the C++ API reference.

```
virtual bool ConvertCellToString(
SaaSDataCell& in_dataCell,
simba_wstring& out_data) = 0;
```

Convert the data cell value to String. Conversion will take place in following steps:

1. Get the data in correct format as per the library used to parse that type of data. For example, JSON data can be stored in `rapidjson::Value`.
2. The data got from step 1 can be converted to wide string by using the methods provided by library used to handle that type of data and by using the methods of `simba_wstring`.
3. The string formed in step 2 to be stored in `out_data` and return true if conversion is successful or false if not successful or data is NULL

```
.virtual void GetDOMMemberValue(

IParsedDoc& in_parsedDoc,

const simba_wstring& in_path,

SaaSDataCell& out_value) = 0;
```

This method is used to get the data from the parsed response (DOM) by providing the data path as input. The path in_path is searched in the parsed response of in_ parseddoc and if data is found it is stored in out_value data cell.

Modify HTTP Request / Response

To get the data using REST API, an HTTP request is formed by SDK based on the input configuration details provided to fetch data in connector and then the request is sent to server to get the intended data. And the server responds with the requested data and finally SDK parses the data which can be viewed in ODBC application.

However, due to wide range of features supported by REST API, it could be a case that the HTTP request formed by SDK based on the configuration provided for example, using available configuration options, it is not formed as intended and needs to be modified before it is sent to Server. In this case, you can leverage the feature of SDK to Modify the request before it is actually sent to server (whether it is an Authentication, Pagination or non-paginated request) so that correct request is sent to server to get the intended data.

You can extend the `IHTTPRequestModifierHandler` interface and implement the abstract method `ModifyRequest`:

```
virtual void ModifyRequest(SaaSRequestData& io_
saasRequestData) = 0;
```

You can modify the parts of the request such as host, port, headers, query parameters, path, body, request type, request timeout. To modify these details, setter methods are available in `SaaSRequestData`. For more details, check `IHttpRequestModifierHandler` and `SaaSRequestData` in C++ API reference.

HTTP response is parsed by parser to get the intended data. But sometimes the response may be such that we cannot form the Path to the intended data directly such that parser is able to parse it correctly. Consider the below example of such case:

```
{
“plans”: [
“OTT”: [
“months”: 3,
“price”: 1500
],
“Calling”: [
“months”: 2,
“price”: 200
].
]
}
```

From this server response, to get the Plans, we cannot directly give the path to data , for example,months, price and the service name. Since the parser would not recognize the path correctly if we give path like (plans.OTT.months, plans.Calling.price). So, this response needs to be modified before parser parses it.

The modified response may look like:

```
{
“plans”: [
{
“Service”: “OTT”,
“months”: 3,
“price”: 1500
},
{
“Service”: “Calling”,
```

```
"months": 2,
"price": 200
}
]
}
```

Now if we have three data columns Service, Months, Price and path to each plans.service, plans.months, plans.price, then the parser parses the data successfully and we get two rows in output.

To modify the response, you can extend the `IHTTPResponseModifierHandler` interface and implement the abstract method `ModifyResponse`:

```
virtual void ModifyResponse(SaaSResponseData& io_
saasRequestData) = 0;
```

You can modify the response code, response headers and response body of the HTTP response. You can get the reference to existing response details (headers, httpcode, responsebody) using the Getter methods of `SaaSResponseData` and then modify it as per your requirement.

For more details, look *`for IHttpResponseModifierHandler, SaaSResponseData`* and `SaaSResponse` in the *Simba REST SDK C++ API reference.*

Connector Configuration Options

Connector Configuration Options lists the configuration options available in the Sample Connector and Custom Connectors alphabetically by key name.

Access Token

Key Name	Default Value	Required
`Auth_AccessToken`	None	Yes, if `Auth_Type` is set to `Access Token`.

Description

The access token for authentication.

Auth_Type

Key Name	Default Value	Required
`Auth_Type`	None	Yes

Description

This option specifies the authentication method to use.

Select from the following:

- To authenticate your connection using your datasource user name and password, set this option to `Basic`.
- To authenticate your connection by going through an OAuth 2.0 authentication flow or by refreshing an expired access token and then using it, set this option to `OAuth_2.0`.
- To authenticate your connection using an already generated access token, set this option to `Access Token`.

Client ID

Key Name	Default Value	Required
`Auth_Client_Id`	None	Yes, if `Auth_Type` is set to `OAuth_2.0`.

Description

The client ID associated with your DataSource application.

Client Secret

Key Name	Default Value	Required
`Auth_Client_Secret`	None	Yes, if `Auth_Type` is set to `OAuth_2.0`.

Description

The client secret associated with your DataSource application.

If both `Auth_Client_Secret` and `EncClientSecret` are set, the connector uses `Auth_Client_Secret`.

ConnIdentifier

Key Name	Default Value	Required
`ConnIdentifier`	Empty string	No

Description

This option is used to specify connection unique identifier to be used to identify the connection. This can be used to configure the Connection level Properties for each individual connection.

DisableColumnPushdown

Key Name	Default Value	Required
DisableColumnPushdown	0	No

Description

This option disables the column pushdown feature for all tables.

- 1: The connector disables the column pushdown feature.
- 0: The connector enables the column pushdown feature.

Important:

If column pushdown is disabled, some columns may not be returned by the datasource server. In this acse, the data in these columns is shown as NULL.

Driver

Key Name	Default Value	Required
Driver	Simba OneDrive ODBC Connector	Yes

Description

The name of the installed connector (Simba OneDrive ODBC Connector).

EnableURLLog

Key Name	Default Value	Required
EnableURLLog	False	No

Description

This option specifies whether the connector logs HTTP request URLs.

- TRUE: HTTP request URLs are logged in the DEBUG channel.
- FALSE: No URL information is logged.

Note:

The log level must be set to DEBUG (5) or TRACE (6) for this information to be captured.

EncClientSecret

Key Name	Default Value	Required
EncClientSecret	None	No

Description

An encrypted version of the client secret for OAuth 2.0 authentication. If both Auth_Client_Secret and EncClientSecret are set, the connector uses Auth_Client_Secret.

EncodingType

Key Name	Default Value	Required
EncodingType	NULL	No

Description

This option specifies the compression encoding method to use. Compressing your data can improve connector performance by improving the rate at which data is passed between the connector and the data store.

- OFF:The data is not compressed.
- NULL: The connector chooses a supported compression method that best compresses the data.

EncryptSwapFile

Key Name	Default Value	Required
EncryptSwapFile	0	No

Description

This option specifies whether the swap file is encrypted.

> **Important:**
>
> Enabling swap file encryption can signifiacntly decrease performance. Only enable swap file encryption in situations where it is required.

- Enabled (`1`): The swap file is encrypted.
- Disabled (`0`): The swap file is not encrypted.

HideSchemas

Key Name	Default Value	Required
`HideSchemas`	None	No

Description

The name of a schema that you want to hide.

When you hide a schema, it is disabled during the connection, and is therefore excluded from SQLTables calls. Hiding a schema can shorten BI tool initialization times, and boost connector performance by shortening the body of each response.

You can set this option to one of the Schema names supported by your Custom Connector.

Host

Key Name	Default Value	Required
`Host`	None	Yes

Description

The URL of the DataSource instance.

LazyInitialization

Key Name	Default Value	Required
`LazyInitialization`	`1`	No

Description

This option specifies whether the connector initializes tables and columns when they are required, or at connection time:

- `1`: The connector initializes tables and columns when they are required.
- `0`: The connector initializes all tables and columns at connection time.

Log Level

Key Name	Default Value	Required
`LogLevel`	`0`	No

Description

Use this property to enable or disable logging in the connector and to specify the amount of detail included in log files.

⚠ Important:

- Only enable logging long enough to capture an issue. Logging decreases performance and can consume a large quantity of disk space.
- When logging with connection strings and DSNs, this option only applies to per-connection logs.

Set the property to one of the following values:

- `0`: Disable all logging.
- `1`: Logs severe error events that lead the connector to abort.
- `2`: Logs error events that might allow the connector to continue running.
- `3`: Logs events that might result in an error if action is not taken.
- `4`: Logs general information that describes the progress of the connector.

- 5: Logs detailed information that is useful for debugging the connector.
- 6: Logs all connector activity.

When logging is enabled, the connector produces a log file named [[[Undefined variable General.CompanyNameShortLC]]][[[Undefined variable General.ProductNameShortLC]]]odbcdriver.log at the location that you specify in the Log Path (LogPath) property.

Log Path

Key Name	Default Value	Required
LogPath	None	Yes, if logging is enabled.

Description

The full path to the folder where the connector saves log files when logging is enabled.

Important:

When logging with connection strings and DSNs, this option only applies to per-connection logs.

Max File Size

Key Name	Default Value	Required
LogFileSize	20971520	No

Description

The maximum size of each log file in bytes. After the maximum file size is reached, the connector creates a new file and continues logging.

If this property is set using the Windows UI, the entered value is converted from megabytes (MB) to bytes before being set.

Important:

When logging with connection strings and DSNs, this option only applies to per-connection logs.

Max Number Files

Key Name	Default Value	Required
LogFileCount	50	No

Description

The maximum number of log files to keep. After the maximum number of log files is reached, each time an additional file is created, the connector deletes the oldest log file.

Important:

When logging with connection strings and DSNs, this option only applies to per-connection logs.

MaxOffset

Key Name	Default Value	Required
MaxOffset	10000	No

Description

This property specifies the maximum page size to retrieve rows.

MemoryManagerMemoryLimit

Key Name	Default Value	Required
MemoryManagerMemoryLimit	1024 for the 32-bit connector, or 2048 for the 64-bit connector.	No

Description

The maximum amount of RAM in megabytes (MB) that the connector can use to cache data for SQL operations.

The data type of the value is String, but the value must be an integer.

MemoryManagerStrategy

Key Name	Default Value	Required
`MemoryManagerStrategy`	`2`	No

Description

This option specifies the following connector behavior for executing SQL operations on non-SQL data:

- How the connector restricts RAM usage for individual SQL operations.
- How the connector responds when SQL operations require more RAM than the allotted amount.
- Whether to maximize the performance of fewer operations or to support more concurrent operations.

The total amount of RAM that can be used by all concurrent operations is always determined by the memory limit (the `MemoryManagerMemoryLimit` setting).

Set the key to one of the following values:

- `1`: Each operation is allotted an amount of RAM based on an internal calculation. If an operation requires more RAM than the allotted amount or if the memory limit is reached, then the connector stops the operation or operations and returns an out-of-memory error. This setting is recommended for situations where the connector should not write to disk, such as in cloud deployments.
- `2`: Each operation is allotted an amount of RAM based on an internal calculation. If an operation requires more RAM than the allotted amount or if the memory limit is reached, then the connector creates swap files on disk and moves some of the cached data to those files in order to free up memory and continue the operation or operations. This setting is recommended if you need to run multiple concurrent queries with good performance.
- `3`: The first operation can consume as much RAM as necessary from the amounts specified by the memory limit and threshold, while subsequent operations use the remaining amount. If the memory limit or threshold is reached, then the connector creates swap files on disk and moves some of the cached data to those files in order to free up memory and continue the operation or operations. This setting is recommended if you need to maximize performance for a single query.

MemoryManagerSwapDiskLimit

Key Name	Default Value	Required
`MemoryManagerSwapDiskLimit`	`0`	No

Description

The maximum total size of the swap files that the connector creates to temporarily cache data on disk, in megabytes (MB).

When this option is set to `0`, there is no limit to the size of the swap files.

MemoryManagerThresholdPercent

Key Name	Default Value	Required
`MemoryManagerThresholdPercent`	`80`	No

Description

The maximum percentage of the memory limit (the `MemoryManagerMemoryLimit` setting) that can be used by an existing SQL operation. The remaining memory limit is reserved for new operations.

The data type of the value is String, but the value must be an integer between `0` and `100`, inclusive.

Minimum TLS

Key Name	Default Value	Required
`Min_TLS`	TLS 1.2 `(1.2)`	No

Description

The minimum version of TLS/SSL that the connector allows the data store to use for encrypting connections. For example, if TLS 1.1 is specified, TLS 1.0 cannot be used to encrypt connections.

- TLS 1.0 (`1.0`): The connection must use at least TLS `1.0`.
- TLS 1.1 (`1.1`): The connection must use at least TLS `1.1`.
- TLS 1.2 (`1.2`): The connection must use at least TLS `1.2`.

MinOffset

Key Name	Default Value	Required
`MinOffset`	`500`	No

Description

This property specifies the minimum page size to retrieve rows.

Password

Key Name	Default Value	Required
`PWD`	None	Yes, if configuring the connector to go through a basic authentication flow.

Description

The password corresponding to the user name that you provided in the User field (the `UID` key).

Proxy Host

Key Name	Default Value	Required
`ProxyHost`	None	Yes, if connecting through a HTTP proxy.

Description

The host name or IP address of a HTTP proxy that you want to connect through.

Proxy Password

Key Name	Default Value	Required
ProxyPwd	None	Yes, if connecting to a HTTP proxy that requires authentication.

Description

The password that you use to access the HTTP proxy.

Proxy Port

Key Name	Default Value	Required
ProxyPort	None	Yes, if connecting through an HTTP proxy.

Description

The number of the port that HTTP proxy uses to listen for client connections.

Proxy Uid

Key Name	Default Value	Required
ProxyUid	None	Yes, if connecting to an HTTP proxy that requires authentication.

Description

The user name that you use to access the HTTP proxy.

RequestTimeout

Key Name	Default Value	Required
RequestTimeout	0	No

Description

The maximum time in seconds that you allow the HTTP transfer operation to take. The minimum value is 1 and there is no maximum value.

> **Important:**
>
> It is normal for name lookups to take a considerable amount of time to process; using this optioj can cause other operations to be aborted. It is recommended that you only use this option in the case of a failing HTTP transfer.

Swap File Path

Key Name	Default Value	Required
`SwapFilePath`	The default temporary directory for your operating system.	No

Description

The full path to the directory where the connector creates swap files to temporarily cache data on disk.

Use HTTPS

Key Name	Default Value	Required
`UseEncryptedEndpoints`	Enabled `1`	No

Description

This option specifies whether the data source endpoints are encrypted using HTTPS.

- Enabled (`1`): The connector uses encrypted endpoints.
- Disabled (`0`): The connector does not use encrypted endpoints.

UseHostVerification

Key Name	Default Value	Required
`UseHostVerification`	Enabled `1`	No

Description

This option specifies whether the connector requires the host name in the server's certificate to match the host name of the server when connecting over SSL.

- Enabled (`1`): The connector requires the host name in the server's certificate to match the host name of the server that you are connecting to.
- Disabled (`0`): The connector accepts all host names.

UsePeerVerification

Key Name	Default Value	Required
`UsePeerVerification`	Enabled `1`	No

Description

This option specifies whether the connector verifies the identity of the server when connecting over SSL.

- Enabled (`1`): The connector verifies the identity of the server.
- Disabled (`0`): The connector does not verify the identity of the server.

User

Key Name	Default Value	Required
`UID`	None	Yes, if configuring the connector to go through a Basic authentication flow.

Description

The user name for your datasource account.

UseThrottling

Key Name	Default Value	Required
`UseThrottling`	False (0)	No

Description

This option is used to specify whether connector supports Throttling. Enable to `1` (`true`) if it is supported.

UseWindowsProxySettings

Key Name	Default Value	Required
`UseWindowsProxySettings`	Disabled `0`	No

Description

This option specifies whether the connector retrieves the proxy host and port through the Internet Options in Internet Explorer, when connecting through an HTTP proxy.

This property is applicable only to Windows.

- Enabled (`1`): The connector retrieves the proxy host and port through Internet Explorer.
- Disabled (`0`): The connector does not retrieve the proxy host and port through Internet Explorer.

Contact Us

If you have difficulty 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 REST SDK version and development platform that you are using.

You can contact Technical Support via the Magnitude Support Community at http://magnitudesoftware.com/online-support/.

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

Third-Party Trademarks

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

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, macOS, Mac OS, 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.