Progress DataDirect Autonomous REST Connector for ODBC

User's Guide

Release 8.0.1

Copyright

Visit the following page online to see Progress Software Corporation's current Product Documentation Copyright Notice/Trademark Legend: https://www.progress.com/legal/documentation-copyright.

Updated: 2025/10/17

Table of Contents

Welcome to the Progress DataDirect Autonomous REST Connector for ODBC 11

What's new in this release?....12
Driver requirements....19
Getting started using prebuilt Model files (Windows)....20
Getting started creating a Model file....24
Installing and setting up the driver (Windows)....29
Installing and setting up the driver (Linux)....31
Connection string examples....34
Version string information....37
- getFileVersionString function....38

Data types....38
Driver specifications....39
Mapping objects to tables....40
- Mapping JSON responses....41
- Mapping XML responses....44
- Mapping CSV responses....47
- Determining the primary key....48

Troubleshooting....50
Contacting Technical Support....50

Tutorials 51

The Example application....51
Power BI (Windows only)....53
Tableau (Windows only)....53
Microsoft Excel (Windows only)....54
Connecting to REST APIs using the Composer user interface....56

Configuring and connecting to data sources....65

Environment settings....66
- Windows environment variables....66
- Linux environment variables....66
- UTF-16 applications on Linux....69

Configuring the relational map....69
Generating a Model file with the Autonomous REST Composer....70
- Sampling REST endpoints....74
- Parameter variables....77

Customizing your schema....78
Configuring custom authentication with the Configuration Manager....78
Reviewing the status of your endpoints....80
Configuring data sources with the Configuration Manager....80
Generating connection strings with the Configuration Manager....82
Using a connection string....82
Additional configuration methods for Linux....83
Configuration through the system information (odbc.ini) file....83
DSN-less connections....87
File data sources....88
Testing connections and queries with the Configuration Manager....89
Password Encryption Tool (UNIX/Linux only)....90
Using a logon dialog box....91
Using IP addresses....91
Authentication....92
Basic authentication....93
AWS credentials authentication....94
Bearer token authentication....95
Digest authentication....96
HTTP header authentication....97
URL parameter authentication....98
OAuth 2.0 authentication....99
Custom authentication 119
TLS/SSL encryption....121
Certificates....121
TLS/SSL server authentication....121
TLS/SSL client authentication....122
Connecting through a proxy server....123
Performance considerations....124

Using the SQL engine server....125

Configuring server mode using the Configuration Manager....126
Stopping the SQL engine server using the Configuration Manager....127
Configuring the SQL engine server using Java options....127
Stopping the SQL engine server....129
Configuring Java logging for the SQL engine server....129

Additional features and functionality 131

Identifiers....131
Parameter metadata support....132

Connection option descriptions....133

Access Key....143

Access Token....144
Array Normalization Threshold....145
Authorization Code....145
Authentication Header....146
Authentication Method....147
Authentication URL Parameter....148
Authorization URI....149
Claims Issuer....149
Claims Subject....150
Client ID....150
Client Secret....151
Create Map....152
Crypto Protocol Version....152
Custom Authentication Parameters....153
Data Source Name....154
Debug Folder....155
Default Query Options....156
Description....156
Enable Login Prompt....157
Encryption Method....158
Extended Options....159
Fetch Size....159
Health Check URI....160
Host Name....161
Host Name In Certificate....161
JSON Root....162
JVM Arguments....163
JVM Classpath....164
JVM Path....165
JWT Certificate Alias....166
JWT Certificate Password....166
JWT Certificate Store....167
Key Password....167
Keystore....168
Keystore Password....169
Log Config File....169
Logoff URI....170
OAuth Client Credentials Mode....170
Password....171
Port Number....172
Proxy Host....172
Proxy Password....173
Proxy Port....173
Proxy User....174
Qualify Normalized Names....174

Read Ahead....175
Redirect URI....176
Refresh Dirty Cache....177
Refresh Schema....177
Refresh Token....178
Region....179
REST Config File....179
REST Sample Path....180
Sampling Failure Tolerance....181
Schema Map....182
Scope....183
Secret Key....183
Security Token....184
Server Port Number....185
Server Proxy Host....185
Server Proxy Password....186
Server Proxy Port....186
Server Proxy User....187
SQL Engine Mode....187
SQL Service....188
Statement Call Limit....188
Statement Call Limit Behavior....189
Table....190
Token URI....190
Transaction Mode....191
Truststore....192
Truststore Password....192
User....193
User Agent String....193
Validate Server Certificate....194
Web Service Fetch Size....195
Web Service Pool Size....196
Web Service Retry Count....197
Web Service Timeout....197

Model file syntax....199

HTTP response code processing....201
OAuth 2.0 authentication....204
Custom authentication requests....206
Table definition entries....209
- Schema name....212
- Write operations....213
- Paging....215
- REST model parsing....223

POST requests....224
Requests with custom HTTP headers....228
Requests with custom parameters....229
Query paths....230
Column names....234
Data type mapping....235
Primary key....237
Read-only columns....237
Nullable columns....238
Columns as an array....238
Columns as a key-value map....239
Columns with nested objects....239
Date, time, and timestamp formats....240
Subfields....241
Columns as HTTP headers....241
Filtering and URI parameters....242
User-defined functions and procedures....244
Parameters....247
Data types syntax....248
DETERMINISTIC statements....248
NULL-input statements....249
SPECIFIC statements....249
Routine language....250
EXTERNAL statements....250
Result sets....251
RETURN statements....251
Compound statements....252
Procedural SQL statements....253
URL-encoded values....264
Example Model file....266

Supported SQL statements and extensions....269

Alter Session (EXT)....269
Delete....270
Insert....271
Select....272
Select clause....274
Update....282
SQL expressions....283
Column names....284
Literals....284
Operators....286
Functions....290
Conditions....290

Subqueries....291
IN predicate....292
EXISTS predicate....292
UNIQUE predicate....292
Correlated subqueries....293

Pre-defined stored procedures 295

Amazon S3 stored procedures....296
copyObject....296
downloadObject 297
uploadObject....297
Box stored procedures....298
copyFile....298
copyFolder....298
downloadObject 299
uploadObject....299
Dropbox stored procedures....300
deleteFile....300
downloadFile....301
uploadFile....301
Google Drive stored procedures....302
downloadObject 302
uploadObject....302
Microsoft Azure Data Lake stored procedures 303
appendFile 304
copyFile 304
createFile....305
createFolder....305
downloadFile....306
flushFile 306
uploadFile....307
writeFileContents....307

1

Welcome to the Progress DataDirect Autonomous REST Connector for ODBC

The Progress DataDirect Autonomous REST Connector for ODBC is a driver that supports SQL read and update access to JSON, XML, and CSV based REST API data sources. To support SQL access to REST services, the driver creates a relational map of the returned data and translates SQL statements to REST API requests. The driver can either infer a map at the beginning of a session or leverage a configuration REST file (Model file) that allows you to modify and persist a map. In addition, the driver employs a SQL engine component that provides support to SQL constructs unavailable to most REST services. This functionality offers a number of advantages, including support for reporting data and metadata in a form ODBC applications are ready to use.

Once you are ready to start accessing your data with your application, the driver requires you to complete several tasks to ensure that the driver deployed properly. The following topics help you get started acquiring a Model file and connecting to your REST service:

- Getting started using prebuilt Model files (Windows) on page 20
- Getting started creating a Model file on page 24
- Installing and setting up the driver (Windows) on page 29
- Installing and setting up the driver (Linux) on page 31

Note: The documentation library uses the terms *the driver* and *the connector* to refer to the Autonomous REST Connector.

The documentation for the driver also includes the *Progress DataDirect for ODBC Drivers Reference*. The reference provides general reference information for all DataDirect drivers for ODBC, including content on troubleshooting, supported SQL escapes, and DataDirect tools. For the complete documentation set, visit to the Progress DataDirect Connectors Documentation Hub:
https://docs.progress.com/bundle/datadirect-connectors/page/DataDirect-Connectors-by-data-source.html.

For details, see the following topics:

- What's new in this release?
- Driver requirements
- Getting started using prebuilt Model files (Windows)
- Getting started creating a Model file
- Installing and setting up the driver (Windows)
- Installing and setting up the driver (Linux)
- Connection string examples
- Version string information
- Data types
- Driver specifications
- Mapping objects to tables
- Troubleshooting
- Contacting Technical Support

What's new in this release?

Visit the following web pages for the latest support and certification information.

- Release Notes: https://www.progress.com/odbc/release-history/
- DataDirect Product Compatibility Guide:
https://docs.progress.com/bundle/datadirect-product-compatibility/resource/datadirect-product-compatibility.pdf

Changes Since 8.0.1 GA

- **Enhancements:**
  - The Qualify Normalized Names option has been added to the driver. This option allows you to configure whether the names of relational tables normalized from array columns are derived directly from the column name or prefixed with parent object names. For details, see Qualify Normalized Names on page 174.
  - The driver has been enhanced to support OAuth 2.0 flows that require client credentials be specified in the body of a POST request. You can configure the driver to send client credentials in the body of a POST request using the new `0` (POST) setting for the OAuth Client Credentials Mode (ClientCredentialsMode) option. See OAuth Client Credentials Mode on page 170 and OAuth 2.0 authentication on page 99 for more details.

- The driver is now compiled with a Visual Studio 2022 compiler for the Windows platforms. As a result, you must have Microsoft Visual C/C++ runtime version 14.40.33810 or higher on your machine to run the driver.
- The following enhancements have been made to improve how the driver handles assigning primary keys for tables:
  - The logic the driver uses to assign the default primary key has been improved to provide more accurate results.
  - A list of viable primary key candidates can be reviewed by querying the `INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS` system table. If you need to designate a primary key other than the default, the `PRIMARY_KEY_CANDIDATES` column provides you with a quick reference of candidates ranked by the how well they meet the primary key criteria.
  - The driver has been enhanced to generate a primary key column, `ROWID`, if no viable candidates were discovered during sampling.

  See Determining the primary key on page 48 for more details.
- The driver has been enhanced to detect JSON roots in endpoint responses during sampling. When mapping the response, the driver maps the embedded objects in the root to a dedicated table. You can modify the detected JSON root for an endpoint using the JSON Root field in the Autonomous REST Composer or the JSON Root (JSONRoot) option. See Sampling REST endpoints on page 74 and JSON Root on page 162 for more details.
- The driver has been enhanced to successfully connect and expose a schema even if some of the endpoints in the Model are unable to be sampled. To support this new behavior, the following new features have been introduced:
  - The driver now supports configurable failure tolerance when sampling endpoints. By configuring the new Sampling Failure Tolerance (SamplingFailureTolerance) option, you can specify the number of endpoints for which sampling can fail before the driver fails the connection. See Sampling Failure Tolerance on page 181 for more details.
  - The driver has been enhanced to allow you to review the status of endpoints to verify that they have been successfully sampled. You can query the statuses of your endpoints in the new `INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS` system table. See Reviewing the status of your endpoints on page 80 for more details.
- The driver has been enhanced to support OAuth 2.0 flows that require client credentials be specified in only a basic authentication header or only as a URL parameter to request an access token. You can configure how client credentials are sent in a request using the OAuth Client Credentials Mode (ClientCredentialsMode) option. See OAuth Client Credentials Mode on page 170 and OAuth 2.0 authentication on page 99 for more details.
- The driver has been enhanced to support specifying default filter values (Where clauses) for SQL queries. You can specify your list of default parameters using the Default Query Options (DefaultQueryOptions) options. See Default Query Options on page 156 for more details.
- The driver's next page token paging support has been enhanced to use URLs, query parameter values, and HTTP header values that are returned in either a response body and header. To configure these mechanisms, the driver has added support for the new `nextPageRequestHeader` and `nextPageResponseHeader` parameters in the Model files. See Paging on page 215 for more details.
- The Configuration Manager has been enhanced to simplify configuring OAuth 2.0 authentication. When specifying one of the new grant flow specific values for the Authentication Method (AuthenticationMethod) option, the Configuration Manager exposes only options related to your grant flow. This functionality assists you in configuring your grant flow by showing you what information the driver needs to authenticate. Note that the existing `OAuth2` value for the Authentication Method option will continue to be supported. See OAuth 2.0 authentication on page 99 and Authentication Method on page 147 for details.

- The driver has been enhanced to the support the JWT (JSON Web Token) bearer grant for OAuth 2.0 authentication. You can configure the JWT bearer grant authentication using the new Claims Issuer, Claims Subject, JWT Certificate Alias, JWT Certificate Password, and JWT Certificate Store options. See JWT bearer grant on page 108 for more details.
- The driver has been enhanced to support the PKCE grant type for OAuth 2.0 authentication. See PKCE grant on page 112 for more details.
- The driver has been enhanced to support next token paging for APIs that use a query parameter (for example, `starting_after`) to determine what data value to start after when returning the next page of results. See Paging on page 215 for details.
- The driver has been enhanced to support issuing POST requests with an empty body. You can enable this functionality using the new `#omitWhenEmpty` parameter in the REST model file. See POST requests with an empty body on page 227 for details.
- The driver has been enhanced to support fetching access and refresh tokens at connection when OAuth2.0 is enabled. When using the new dynamic authorization code grant, you can initiate an authorization code grant flow by specifying login credentials using the login prompt for your REST service, thereby providing a method to authenticate without fetching access and refresh tokens via the Configuration Manager or third-party application. In addition, the new Enable Login Prompt (EnableLoginPrompt) option has been added to configure this functionality. See Dynamic authorization code grant on page 105 and Enable Login Prompt on page 157 for details.
- The driver has been enhanced to support modifying documents in document stores using pre-defined stored procedures. The prebuilt Model files for the following data stores have been updated to allow your applications to fetch, insert, update, and delete documents:
  - Amazon S3
  - Box
  - Dropbox
  - Google Drive
  - Microsoft Azure Data Lake Storage

  See Pre-defined stored procedures on page 295 for a list of supported procedures.
- The driver has been enhanced to support user-defined functions and procedures. To be used by the driver, user-defined functions and procedures can be defined in the Model file or by the application. See User-defined functions and procedures on page 244 for more supported functionality and syntax.
- Update operation functionality has been enhanced to support APIs that require using PATCH or PUT methods that send only changed fields in the POST body. You can configure update operations using the `#update` and `#sendOnlyUpdated` parameters in the Model file. See Write operations on page 213 for more information.
- The driver has been enhanced to support designating columns as read-only using the model file with the new `#readOnly` element. See Read-only columns on page 237 for more information.
- The driver has been enhanced to support flagging columns as nullable using the model file with the new `#notNull` element. See Nullable columns on page 238 for more information.
- The driver has been enhanced to support write operations, including Insert, Update, and Delete statements. To enable write operations for an endpoint, you must configure the new `#insert`, `#update`, `#delete` parameters in the Model file. See Write operations on page 213 and Supported SQL statements and extensions on page 269 for more information.
- The Autonomous REST Composer has been enhanced to allow you to define parameter values in endpoints as variables. This functionality simplifies the sharing of Model files by allowing you to designate user-specific values as variables. Users with whom you share the file can then provide default values

for these variables that are replaced across their REST model file, allowing them to quickly customize the endpoints according to their use cases. See Parameter variables on page 77 for more information.

- The driver has been updated with the new JSON Root (JSONRoot) connection option, which allows you to limit the results mapped to tables to only the specified object when you have multiple objects in an endpoint. This option provides a method to map only the data relevant to your application. See JSON Root on page 162 for more information.
- The Autonomous REST Composer has been enhanced to allow you to limit the values displayed in the Authentication Method drop-down field to only those supported by the data source, instead of all those supported by the driver. You can select which fields you want to display by selecting them from the Configure Authentication Method(s) field on the Connection tab. See Generating a Model file with the Autonomous REST Composer on page 70 for more information.

- **Changed behavior:**
  - The behavior of the OAuth Client Credentials Mode (ClientCredentialsMode) option has been updated:
    - The behavior of the `0` setting has been changed so that client credentials are sent only as a basic authentication header. For earlier builds of the driver, client credentials were sent as both a basic authentication header and a URL parameter.
    - The `0` (All) setting has been changed to `0` (Default) in the Configuration Manager and documentation to reflect the new behavior.
    - The default setting for the OAuth Client Credentials Mode is `0` (Default).

    See OAuth Client Credentials Mode on page 170 for more supported functionality and syntax.
  - The Autonomous REST Composer tool is now completely packaged in the driver jar file (`autorest.jar`). As a result, the following files associated with launching the Composer are no longer installed with the product:
    - `ArcUILauncher.exe`
    - `ivcurl28.dll` (32-bit) and `ddcurl28.dll` (64-bit)
  - Terminology changes in the product interface and documentation:
    - The input REST files and Recipe files are now collectively referred to as Model files. The functionality of the files has not been modified as a result of this change.
    - The REST Management Tool is now referred to as the Autonomous REST Composer. The functionality of the tool has not been modified as a result of this change.

Changes for 8.0.1 GA

- **Enhancements:**
  - The driver has been enhanced to support calculating Base64 encoding for user name and password values. When using custom authentication, you can enable Base64 encoding with the Model file. The driver then encodes the values of the user name and password properties when authenticating to your services. For details, see Custom authentication requests on page 206.
  - On Windows, the driver now includes the Configuration Manager for quick configuration and testing of your driver. This Configuration Manager allows you to:
    - Configure data sources
    - Generate and edit connection strings

  - Test connect data sources and connection strings
  - Execute SQL commands for testing
  - Fetch OAuth tokens and configure OAuth
  - Access connection option descriptions and the full product documentation

- The DataDirect ODBC Configuration Manager has been enhanced to support the generation of Model files. This provides you with a method to quickly generate and edit a REST input file. See Generating a Model file with the Autonomous REST Composer on page 70 for details.
- The driver now includes a library of Progress developed Model files to connect to publicly accessible REST services. The prebuilt Model files define the requests and pagination settings for a data source, eliminating the need to create your own Model file. After selecting your data source from the Configuration Manager, you only need to provide your authentication credentials to begin accessing your data. See Getting started using prebuilt Model files (Windows) on page 20 for details.
- The Configuration Manager has been enhanced with role-specific work flows:
  - The developer view of the Configuration Manager, Autonomous REST Composer, provides access to the new prebuilt Models library and configuration properties. In addition, the Hub window has been added that includes access to training videos, documentation, and technical support. The developer's view can be launched through the new desktop and Start menu icons.
  - The user's view provides a simplified interface that allows you to configure and test your connection. You can launch the user's view through the ODBC Administrator.

- The driver now supports responses returned in XML and CSV formats in addition to JSON. When sampling an endpoint, the driver detects the format of the response before mapping the objects to the relational view of the data. If multiple formats are supported by the service, the driver defaults to using JSON; however, you can also configure the driver to use your preferred format. See Mapping objects to tables on page 40 for details.
- The driver has been enhanced to support passing custom HTTP headers when using OAuth 2.0 authentication. When OAuth 2.0 is enabled (`AuthenticationMethod=OAuth2`), you can now pass the HTTP header name with the Authentication Header (AuthHeader) option and the ID value with the Security Token (SecurityToken) option. This functionality can be used for passing the ID string for tenant ID authentication. See OAuth 2.0 authentication on page 99 for details.
- The driver has been enhanced to support AWS (Amazon Web Services) credentials authentication. When AWS credentials authentication is enabled (`AuthenticationMethod=43`), you can configure AWS credentials using the new Access Key (AccessKey), Region (Region), and Secret Key (SecretKey) options. See AWS credentials authentication on page 94 for details.
- The driver has been enhanced to support issuing POST requests that use custom parameters. This allows for filtering in scenarios where complex parameter syntax is employed, such as using complicated JSON data or empty arrays. See POST requests on page 224 for details.
- The driver has been enhanced to support issuing GET requests that use custom parameters, such as those supported by JQL or SOQL, when filtering results. Using the custom parameters supported by your service allows queries to be processed before returning results to driver, thereby resulting in more efficient processing. See Requests with custom HTTP headers on page 228 for details.
- The driver has been enhanced to support bearer token and digest authentication. See Bearer token authentication on page 95 and Digest authentication on page 96 for more details.
- The new Health Check URI (`HealthURI`) connection option provides a method to test connectivity for authentication methods, such as Basic, Digest, URL Parameter-based, or HTTP header-based, that do not perform an explicit action upon connection. See Health Check URI on page 160 for details.

- The driver has been enhanced with the new Array Normalization Threshold (ArrayNormalizationThreshold) connection option. Array Normalization Threshold allows you to specify the length of arrays (in elements) at which the driver begins to normalize arrays to child tables when generating a flattened view. This provides you with a method to control the size and focus of your parent table when encountering large arrays or nested arrays. See Array Normalization Threshold on page 145 for details.
- A Password Encryption Tool, called ddencpwd, is now included with the product package. It encrypts passwords for secure handling in connection strings and odbc.ini files. At connection, the driver decrypts these passwords and passes them to the data source as required. See Password Encryption Tool (UNIX/Linux only) on page 90 for details.
- The driver has been enhanced with the new User Agent String (UserAgent) connection option. User Agent String allows you to specify the value of the User-Agent header to be used in HTTP requests. This provides a method to override the default value of the User-Agent header when required by a service. See User Agent String on page 193 for details.
- The driver has been enhanced to support connecting through a proxy server. You can configure this feature using the new Proxy Host (`ProxyHost`), Proxy Port (`ProxyPort`), Proxy User (`ProxyUser`), and Proxy Password (`ProxyPassword`) connection options. See Connection option descriptions on page 133 for details.
- The new Encryption Method (`EncryptionMethod`) connection option allows you to determine when TLS/SSL data encryption is enabled. See Encryption Method on page 158 for details.
- The new Port Number (`PortNumber`) connection option allows you to specify the port number of the server listener the TCP port of the server that is listening for REST API requests. See Port Number on page 172 for details.
- The new Refresh Dirty Cache (RefreshDirtyCache) connection option allows you to determine whether the driver refreshes a dirty cache on the next fetch operation from the cache. See Refresh Dirty Cache on page 177 for details.
- The new Refresh Schema (RefreshSchema) connection option allows you to determine whether the driver automatically refreshes the relational map of when connecting. See Refresh Schema on page 177 for details.
- The new Authorization URI (`AuthURI`) connection option allows you to specify the endpoint used for obtaining an authorization code from a third-party authorization service for OAuth 2.0 implementations. See Authorization URI on page 149 for details.
- The new Statement Call Limit (`StmtCallLimit`) and Statement Call Limit Behavior (`StmtCallLimitBehavior`) connection options allow you to limit the number of calls made to your web service. See Statement Call Limit on page 188 and Statement Call Limit Behavior on page 189 for details.
- The new Web Service Fetch Size (`WSFetchSize`) allows you to specify the number of rows of data the driver attempts to fetch for each JDBC call. This provides you with a method to tune the driver for your ideal balance throughput and response time. See Web Service Fetch Size on page 195 for details.
- The new Web Service Pool Size (`WSPoolSize`) specifies the maximum number of sessions the driver uses. This allows the driver to have multiple web service requests active when multiple JDBC connections are open, thereby improving throughput and performance. See Web Service Pool Size on page 196 for details.
- The new Web Service Retry Count (`WSRetryCount`) connection options specifies the number of times the driver retries a timed-out Select request.
- The new Web Service Timeout (`WSTimeout`) connection option specifies the time, in seconds, that the driver waits for a response to a web service request. See Web Service Timeout on page 197 for details.

- The driver has been enhanced to support the following new paging parameters in the Model file: fieldListParameter, hasMoreElement, pageSizeElement, totalPagesElement, and totalRowsElement. See Paging on page 215 for details.
- The driver has been enhanced to allow you to define custom authentication requests, including the new Custom Auth Params (CustomAuthParams) connection option. If your service does not support one of the standard authentication methods supported by the driver, you can modify the Model file to define token-based authentication flows. See Paging on page 215 for details.
- The driver has been enhanced to allow you to customize how HTTP response status codes are processed by the driver. By configuring the Model file, you can define error responses for codes that are returned by the service, including driver actions and error messages. See HTTP response code processing on page 201 for details.
- The driver has been enhanced to support OAuth 2.0 authentication. See OAuth 2.0 authentication on page 99 for details.
- The driver has been enhanced to support requests for endpoints that use custom HTTP-headers. See Requests with custom HTTP headers on page 228 for details.

- **Changed behavior:**
  - The attribute for User connection option has changed from `LogonID` to `User`.
  - The behaviors assigned to valid values for the Create Map (`CreateMap`) have been modified. See Create Map on page 152 for details.
  - The supported valid values for the Authentication Method (AuthenticationMethod) option have changed from enum to numeric types. If you are using a connection string or `odbc.ini` file for an earlier version of the driver, you will need to update the values specified for these options. See Authentication Method on page 147 for details.
  - The Application Using Threads (`ApplicationUsingThreads`) connection option is no longer supported. The driver now works with single-threaded and multi-threaded applications for all connections.
  - The Result Memory Size (`ResultMemorySize`) connection option is no longer supported. The size of an intermediate result set is now determined by a percentage of the max Java heap size.
  - The IANAAppCodePage (`IANAAppCodePage`) connection option is longer supported. IANAAppCodePage specified Internet Assigned Numbers Authority (IANA) value if your application was not Unicode enabled or if your database character set was not Unicode.
  - The Login Timeout (`LoginTimeout`) is no longer supported. As a result, connection requests do not timeout, but the driver responds to the `SQL_ATTR_LOGIN_TIMEOUT` attribute.
  - The Report Codepage Conversion Errors (`ReportCodepageConversionErrors`) is no longer supported. As a result, the driver behavior has changed to substitute 0x1A for each character that cannot be converted and does not return a warning or error.

Highlights of the 8.0.0 Release

- The driver supports SQL read-only access to REST API endpoints that return JSON payloads. See Supported SQL statements and extensions on page 269 for details.
- The driver supports all ODBC Core and Level 1 functions and some Level 1 and Level 2 features. See Driver specifications on page 39 for details.
- The driver supports standard JSON data types and additional data types through data type inference. See Data types on page 38 for details.
- The driver supports using internal memory or a configurable Model file to define REST responses and relational mapping. See Configuring the relational map on page 69 for details.

- The driver heuristically maps data types, eliminating the need to define native data types in most scenarios. See Data types on page 38 for more information about data type mapping.
- The driver supports basic, HTTP-header based, URL-Parameter based and no authentication. See Authentication on page 92 for details.
- The driver supports the handling of large result sets with configurable paging and the Fetch Size on page 159 connection option. See Configuring the relational map on page 69 for more information on configuring paging.
- The driver includes a new Tableau data source file (Windows only) that provides improved functionality when accessing your data with Tableau. See Tableau (Windows only) on page 53 for more information.

Driver requirements

Data source and platform requirements

For the latest support information, visit the DataDirect Product Compatibility Guide: https://docs.progress.com/bundle/datadirect-product-compatibility/resource/datadirect-product-compatibility.pdf.

Java requirements

- The driver requires a Java Virtual Machine (JVM) that is Java SE 8 or higher. JVM support includes Oracle JDK, OpenJDK, and IBM SDK (Java) distributions.
- For 32-bit drivers, a 32-bit Java Virtual Machine (JVM) is required. For 64-bit drivers, a 64-bit Java Virtual Machine (JVM) is required.
- For Windows, you must set the PATH environment variable to the directory containing the `jvm.dll` for your JVM.
- For Linux, you must set the library path environment variable of your operating system to the directory containing your JVM's `libjvm.so` file and that directory's parent directory.

Windows requirements for 32-bit drivers

- All required network software that is supplied by your database system vendors must be 32-bit compliant.
- You must have Microsoft Visual C/C++ runtime version 14.40.33810 or higher.
- You must have ODBC header files to compile your application. For example, Microsoft Visual Studio includes these files.

Windows requirements for 64-bit drivers

- All required network software that is supplied by your database system vendors must be 64-bit compliant.
- You must have Microsoft Visual C/C++ runtime version 14.40.33810 or higher.
- You must have ODBC header files to compile your application. For example, Microsoft Visual Studio includes these files.

Linux requirements for 32-bit drivers

- If your application was built with 32-bit system libraries, you must use 32-bit drivers. The database to which you are connecting can be either 32-bit or 64-bit enabled.
- An application compatible with components that were built using g++ GNU project C++ Compiler version 3.4.6 and the Linux native pthread threading model (Linuxthreads).

Linux requirements for 64-bit drivers

- An application compatible with components that were built using g++ GNU project C++ Compiler version 3.4 and the Linux native pthread threading model (Linuxthreads).

Getting started using prebuilt Model files (Windows)

This section provides you with an overview of the steps required to install, set-up, and begin accessing data using prebuilt model files developed by Progress. From the Autonomous REST Composer, you have access to a library of prebuilt Model files for publicly available data sources that fully define the required requests and pagination parameters. After downloading a Model, you only need to provide your authentication credentials to begin accessing data.

To begin accessing data with the driver:

1. Install the driver:
   a) After downloading the product, unzip the installer files to a temporary directory.
   b) From the installer directory, run the appropriate installer file to start the installer. The installer file takes the following form:

   `PROGRESS_DATADIRECT_ODBC_`*`nn`*`_WIN_`*`xx`*`_INSTALL.exe`

   c) Follow the prompts to complete installation.

   Note:

   The installer program supports multiple installation methods, including command-line and silent installations. For detailed instructions, refer to the *Progress DataDirect for ODBC Drivers Installation Guide*.

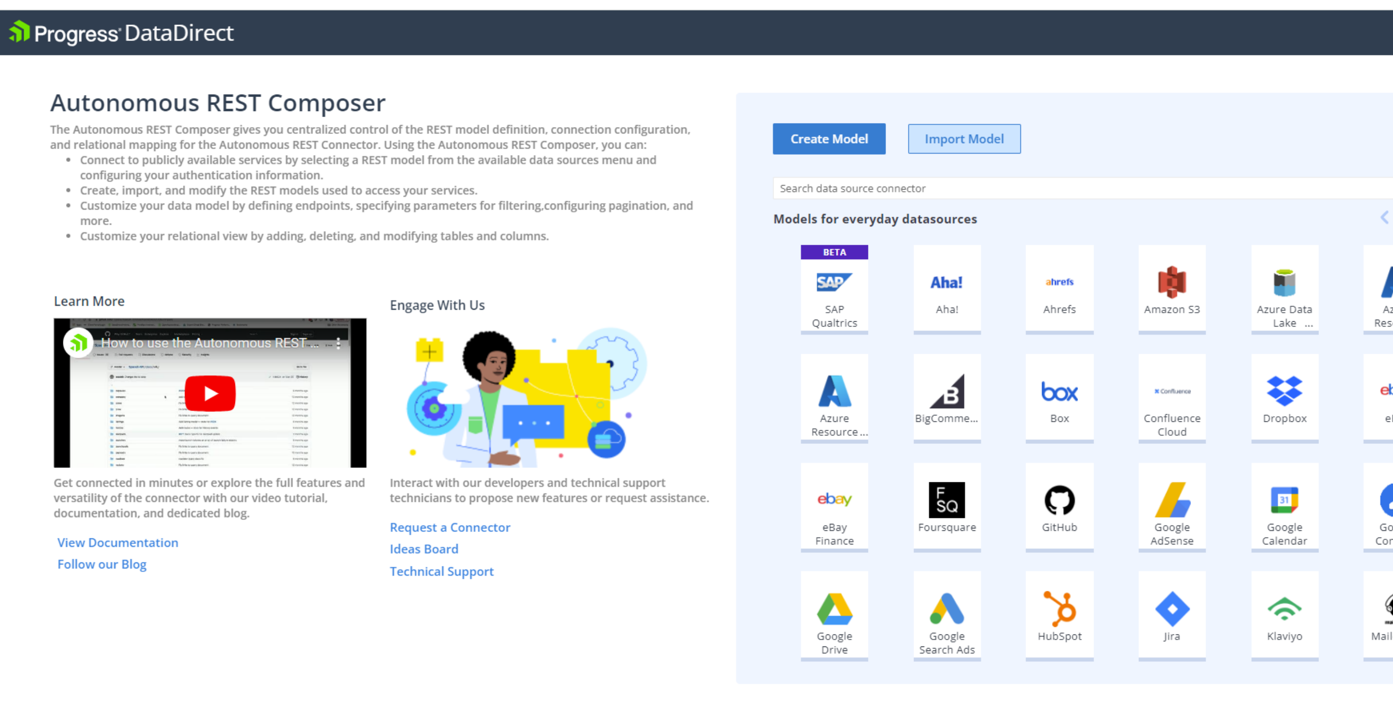
   ---

2. Before you can use your driver, you must set the PATH environment variable to include the path of the `jvm.dll` file of your Java™ Virtual Machine (JVM).
3. Open the Autonomous REST Composer by using one of the following methods:
   - Select the **Autonomous REST Composer (ODBC)** icon from your desktop or the Windows Start menu.
   - From a command line, navigate to the directory containing the `autorest.jar` file and execute the following command:

     ```
     java -jar autorest.jar --odbcdesign
     ```

     By default, the `autorest.jar` file is stored in the following directory: *`install_dir`*`\Progress\DataDirect\ODBC\java\lib\`.

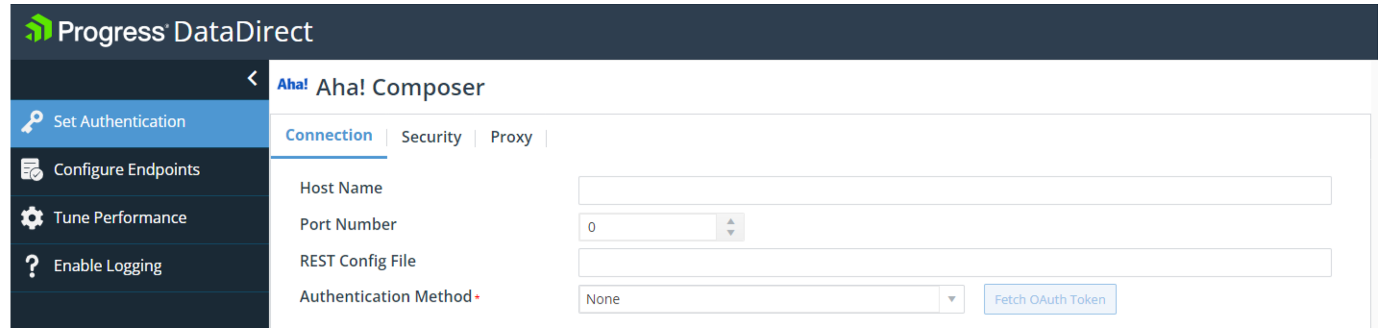
   The Autonomous REST Composer opens in your default web browser.



4. Select the Model for the data source to which you want to connect from the menu on the right.

Note: If you do not see your data source on the menu, you can build your own by following the steps in Generating a Model file with the Autonomous REST Composer on page 70.

5. The **Connection** tab of the Autonomous REST Composer opens.



Optionally, you can provide values for the following fields if you want to connect and view your model in the composer:

- **Host Name**: Set this property to specify the host name portion of the HTTP endpoint to which you send requests. For example, a Jira endpoint would take the following form: `https://`*`mycompany`*`.atlassian.net`.

  Note: This value will be provided by the Model for data sources that have static host names.

- **Port Number**: Optionally, specify the TCP port of the server that is listening for REST API requests.
- **Authentication Method**: Select the authentication method used to connect to your data source. The properties related to the selected method are exposed. Provide values for the exposed fields that apply to your connection. Note that some of the fields may not apply to your data source. Refer to the documentation for your REST service for details.

6. Download your model file:

   a) Select the **Configure Endpoints** tab.

   Important: If you do not provide connection information to your service, you might receive an error message. However, this does not prevent you from downloading your Model file.

   ---

   b) Optionally, add, remove, or edit endpoints that are used to return data for use with SQL-based applications.

   c) Click **Download** to save your model file.

   Note: When configuring a data source or connection string, you will need to specify the fully qualified path to your Model file using the REST Config File (RESTConfigFile) option.

   ---

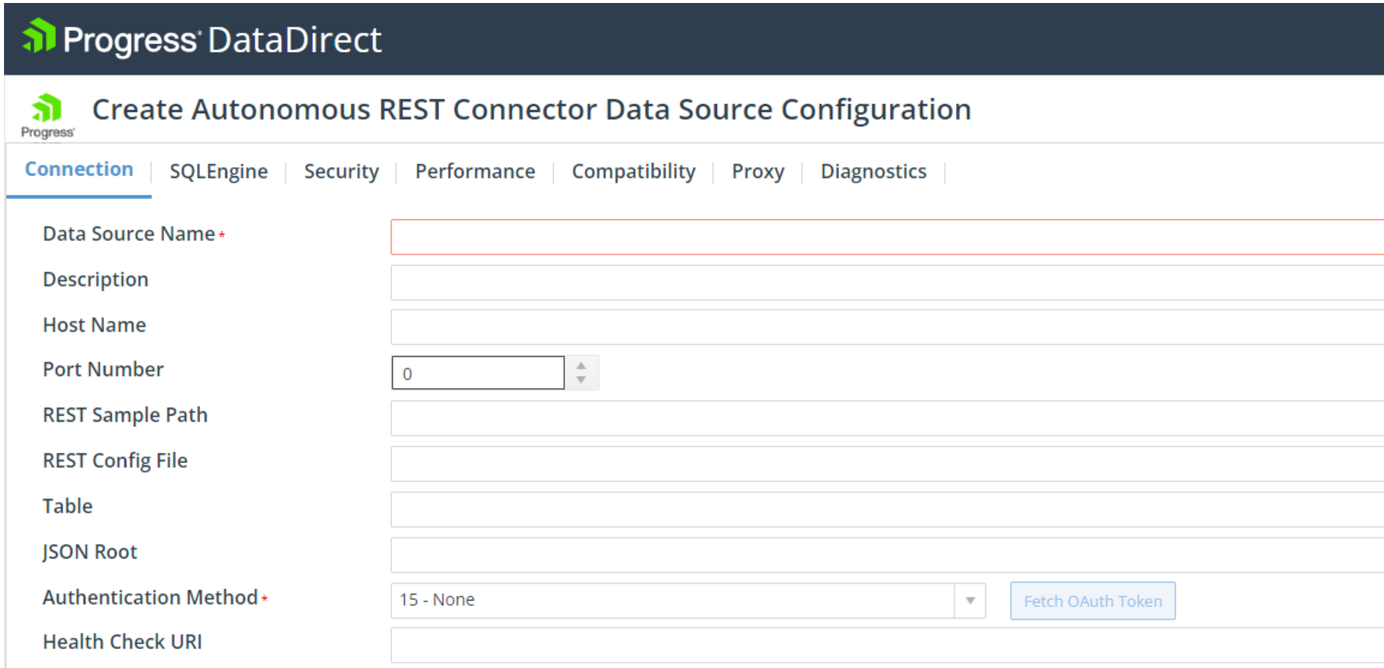
7. Close the Autonomous REST Composer.
8. Start the ODBC Administrator from the Progress DataDirect program group to configure the driver using the ODBC Administrator (GUI). The GUI dialog allows you to configure the data source definitions in the Windows Registry or generate connection strings.

   Note: The Windows driver also supports using connection strings to connect to your service. For more information, see "Using a connection string."

   ---

9. Select either the **User DSN**, **System DSN**, or **File DSN** tab to display a list of data sources.
   - **User DSN**: If you installed a default DataDirect ODBC user data source as part of the installation, select the appropriate data source name and click **Configure** to display the driver Setup dialog box.

     If you are configuring a new user data source, click **Add** to display a list of installed drivers. Select your driver and click **Finish** to display the driver Setup dialog box.
   - **System DSN**: To configure a new system data source, click **Add** to display a list of installed drivers. Select your driver and click **Finish** to display the driver Setup dialog box.
10. The **Connection** tab of the Configuration Manager opens.



Provide values for the following fields:

- **Data Source Name**: Type a string that identifies this data source configuration, such as `Projects`.
- **Description**: Type an optional long description of a data source name, such as `My Development Projects`.
- **Host Name**: Optionally, set this property to specify the host name portion of the HTTP endpoint to which you send requests. For example, a Jira endpoint would take the following form: `https://`*`mycompany`*`.atlassian.net`.

  Note: This value will be provided by the Model for data sources that have static host names.

- **Port Number**: Optionally, specify the TCP port of the server that is listening for REST API requests.
- **REST Config File**: Enter the name and location of your Model file. For example, `C:\path\to\myrest.rest`.

11. From the **Authentication Method** drop-down, select the authentication method used to connect to your data source. The properties related to the selected method are exposed. See Authentication on page 92 for more information on supported methods.
12. Provide values for the exposed authentication fields that apply to your connection. Note that some of the fields may not apply to your data source. Refer to the documentation for your REST service for details.
13. Set the values for any optional properties that you want to configure. For additional information on optional features and functionality, see the following resources:

    - Connection URL examples provides connection string examples that can be used to configure common functionality and features. You can modify and combine these examples to create a string that best suites your environment.
    - Connection property descriptions provides a complete list of supported properties by functionality.

14. To test your configuration:
    a) Click **Test Connect**.
    b) In the **Test Query** field, specify a query that you would like to test.

    Note: The default query returns only a list of the tables defined in the Model file. This list is generated by the driver's internal SQL Engine. To authenticate to the REST service, specify a simple SELECT statement using of the table from the results.

    c) Click **Execute**.

    If successful, the driver will return results from a test query.
15. Click **Save** to save your data source.

    Note: As you provide values for properties, the Configuration Manager also generates an ODBC connection string for use by your application. To use your string, click the Copy button ( ) and paste the string to a location that can be used by your application.
16. Begin accessing data with your applications, BI tools, database tools, and more. To help you get started, the following resources guide you through accessing data with some common tools:
    - Example Application: The example application allows you to test connect and execute SQL statements right out of the box.
    - Power BI: Power BI is a business intelligence software program that allows you to generate analytics and visualized representations of your data.
    - Tableau: Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data.
    - Microsoft Excel: Excel is a spreadsheet tool that allows you to connect, view tables, and execute SQL statements against your data.
    - Supported SQL statements and extensions: This section describes the syntax used for SQL statements supported by the driver. You can modify and use the provided examples for your application or tool.

This completes the deployment of the driver.

Getting started creating a Model file

This section provides you with an overview of the steps required to install the Autonomous REST Connector and create a Model file to be used with your connections. The REST model file defines endpoints and table mapping used to generate your relational model. In addition, the Model file is capable of configuring a number of driver behaviors, such as paging, custom authentication, and HTTP response code processing.

Note: If you are accessing a publicly available data source, refer to the library of Model files for your data source. These Model files contain fully defined requests and pagination setting, allowing you to connect after providing your authentication credentials. See Getting started using prebuilt Model files (Windows) on page 20 for details.

To begin accessing data with the driver:

1. Install the driver:
   a) After downloading the product, unzip the installer files to a temporary directory.
   b) From the installer directory, run the appropriate installer file to start the installer. The installer file takes the following form:

      PROGRESS_DATADIRECT_ODBC_*nn*_WIN_*xx*_INSTALL.exe

   c) Follow the prompts to complete installation.

Note:

The installer program supports multiple installation methods, including command-line and silent installations. For detailed instructions, refer to the *Progress DataDirect for ODBC Drivers Installation Guide*.

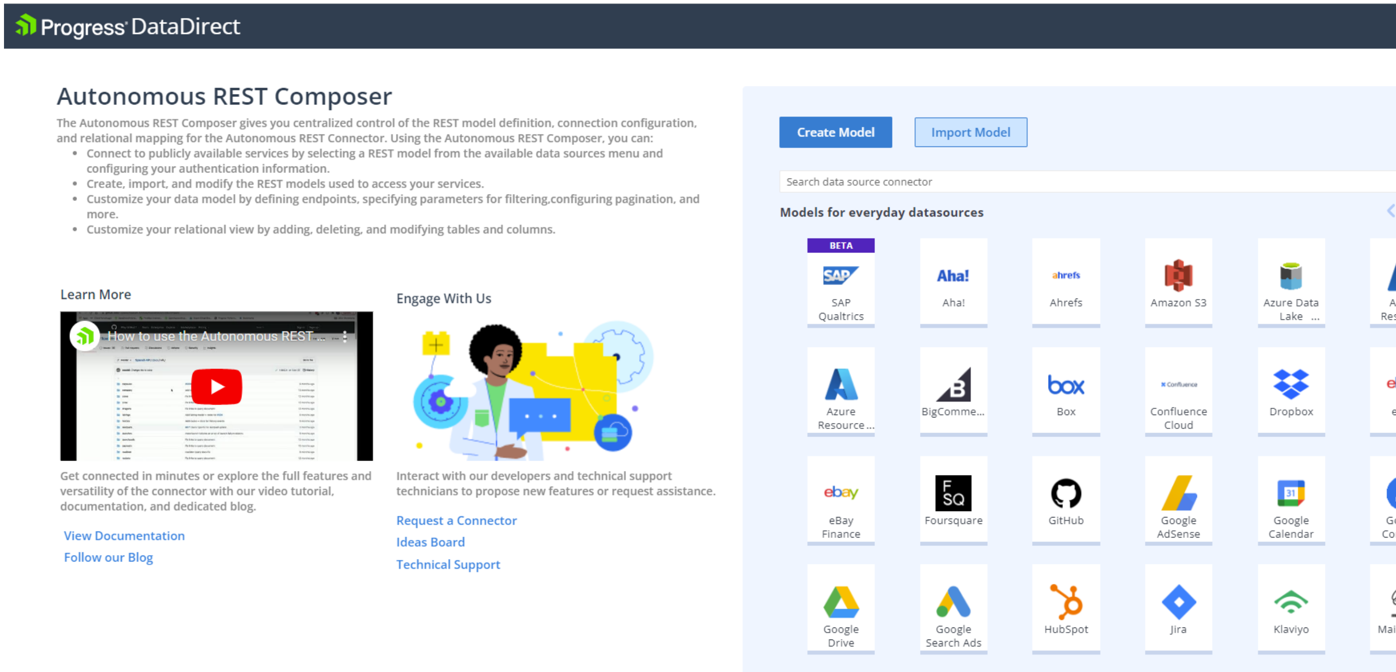
---

2. Before you can use your driver, you must set the PATH environment variable to include the path of the `jvm.dll` file of your Java™ Virtual Machine (JVM).
3. Open the Autonomous REST Composer by using one of the following methods:
   - Select the **Autonomous REST Composer (ODBC)** icon from your desktop or the Windows Start menu.
   - From a command line, navigate to the directory containing the `autorest.jar` file and execute the following command:

     ```
     java -jar autorest.jar --odbcdesign
     ```

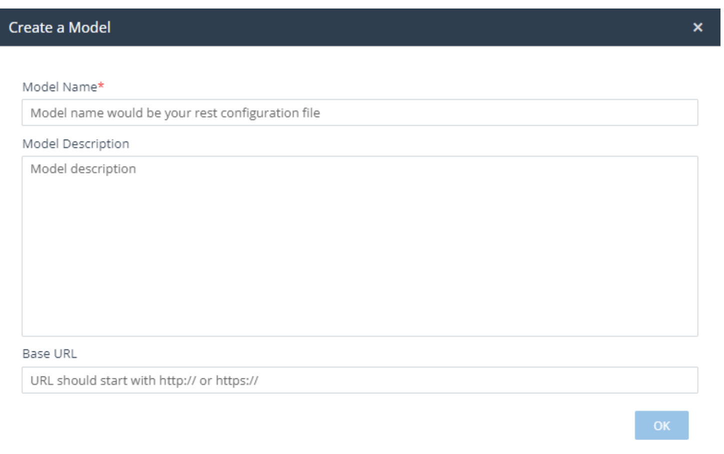
     By default, the `autorest.jar` file is stored in the following directory: *install_dir*\Progress\DataDirect\ODBC\java\lib\.

   The Autonomous REST Composer opens in your default web browser.



4. Select **Create Model**.
5. The **Create Model** window opens.

Figure 1: The Create Model Window

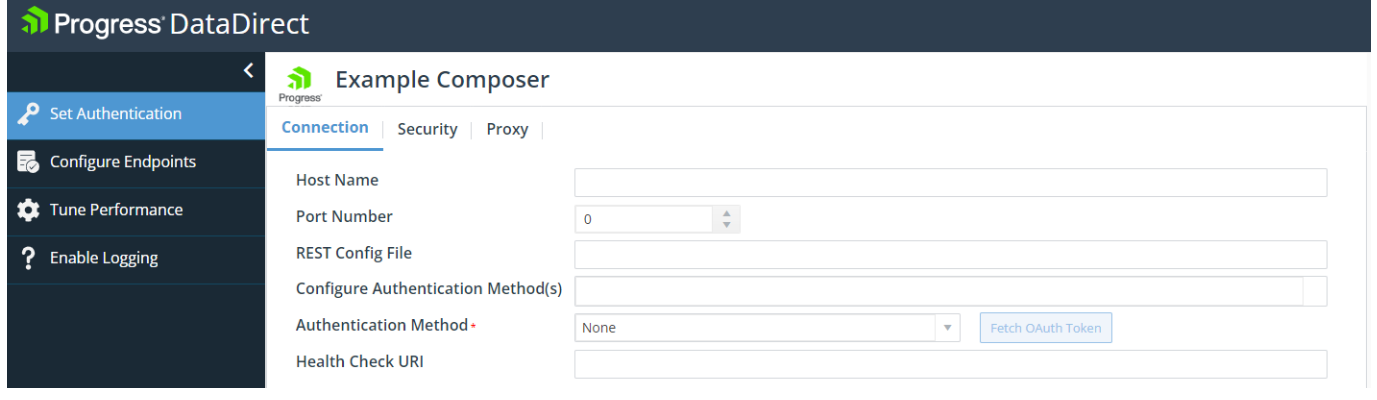


Complete the following fields to create a new project; then, click **OK**:

- **Model Name**: The name of your Model file to be created.
- **Model Description**: An optional description of your Model. Note that this description will be stored in clear text in the Model file.
- **Base URL**: The host name portion of your REST endpoints.

6. The **Connection** tab of the Autonomous REST Composer opens.

Figure 2: The Connection tab



Optionally, provide values for the following fields:

- **Host Name**: Set this property to specify the host name portion of the HTTP endpoint to which you send requests. For example, a Jira endpoint would take the following form: `https://`*`mycompany`*`.atlassian.net`.
- **Port Number**: Optionally, specify the TCP port of the server that is listening for REST API requests.

7. Select the Authentication Method used by your endpoints; then, provide values for the appropriate fields:

- In the Configure Authentication Method(s) field, select the method(s) used by your service.
- In the Authentication Method drop-down, select the method you want to configure for this session. The fields associated with the method you select are exposed.

- In the exposed fields, provide values for the applicable fields. Note that not all of the fields exposed are required for all services.

See Authentication on page 92 for a full description of these methods.

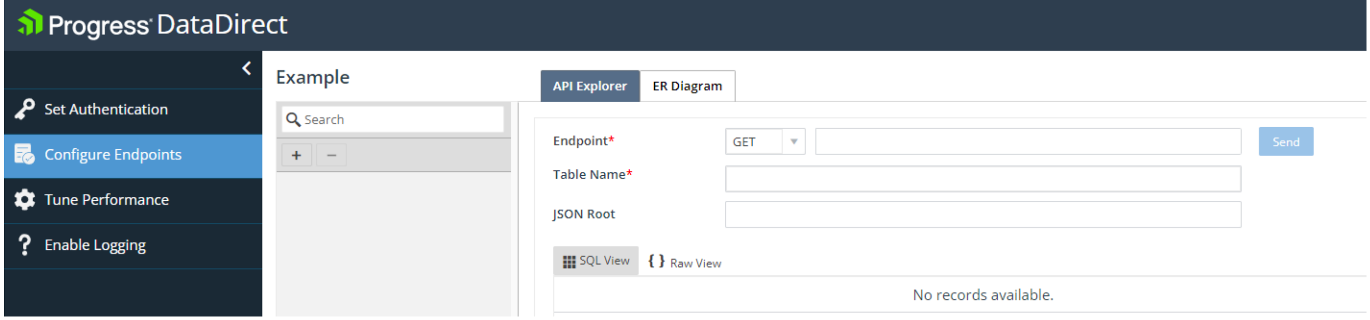
Note: Authentication properties specified through the Configuration Manager are not persisted in the Model file. To share authentication settings among all connections using the file, you must manually update the Model file. See OAuth 2.0 authentication on page 99 for details.

Note: Properties for custom authentication are specified in the Model file. You can update the file manually or using the Configuration Manager. For details, see Configuring custom authentication with the Configuration Manager on page 78.

---

8. Select the **Configure Endpoints** tab on the side menu.

Figure 3: The Configure Endpoints tab

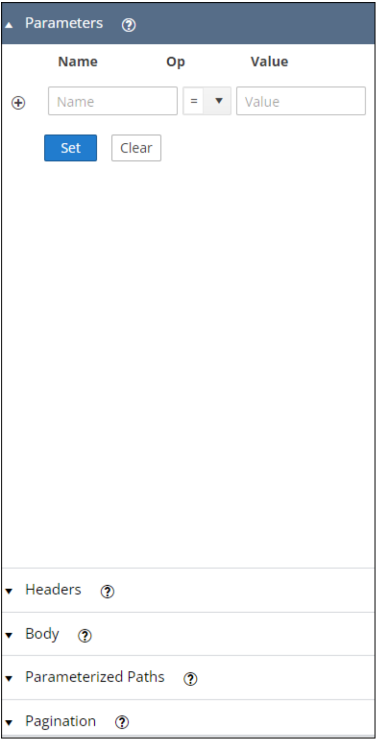


Provide the minimum required information for an endpoint to which you want to issue requests:

- From the **Endpoint** drop-down menu, select the type of request to issue against your endpoint.
- In the **Endpoint** field, type the path portion of your endpoint after the base URL. Note that the value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.
- In the **Table Name** field, type the name of the relational table to which you want the endpoint to map.

9. Optionally, further define your endpoint using the customization pane on the right. For example, specifying query parameters, parameterized paths, and POST request bodies. For detailed descriptions of defining different types of endpoints, see Sampling REST endpoints on page 74.

Figure 4: Customization Pane



10. Click **Send**. The driver sends the REST request and generates a relational view of the data based on the response. To add additional endpoints, click **+** in the request pane on the left.
11. Optionally, in the **Pagination** section of the customization pane, select the paging method to be used for your Model; then, provide values for the applicable paging parameters. For a description of these parameters, see Paging on page 215.
12. Optionally, customize your relational schema, including modifying column names, data type mapping, and primary key designation. See Customizing your schema on page 78 for details.
13. Optionally, click **Test Connect** to test your model by executing SQL queries.
14. Optionally, you can review the status of your endpoints by querying the `SYSTEM_SAMPLING_STATUS` system table. For example:

    ```
    SELECT * FROM INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS
    ```

    This will allow you to verify that your endpoints have been successfully sampled by the driver and diagnose any issues, should they occur. See Reviewing the status of your endpoints on page 80 for details.
15. Click **Download** to generate and download your Model file.
16. Move your Model file to a location to be used by the driver. When configuring your connection string or data source, you will also need to specify this location using the REST Config File (`Config`) connection option.

After creating your Model file, you are ready to configure and connect. See the following to configure your driver for connection:

- Installing and setting up the driver (Windows) on page 29
- Installing and setting up the driver (Linux) on page 31

You can edit your Model file later by opening it by clicking **Import Model** when starting the Autonomous REST Composer.

Note: The Model file generated by the Configuration Manager supports most of the request types and functionality typically used to access a service. However, the driver supports additional features and functionality that are not currently available through the file generated by the Configuration Manager. If you need to configure features or functionality not supported through the Configuration Manager, you can manually edit your generated file using a text editor. See "Model file syntax" for details.

Installing and setting up the driver (Windows)

Before you begin: You will need a Model file before beginning this procedure. You can get one from an administrator or by creating your own (see Getting started creating a Model file on page 24). If you are connecting to a publicly available REST service, you can also use a pre-built Model file (see Getting started using prebuilt Model files (Windows) on page 20).

This section provides you with an overview of the steps required to install, set-up, and begin accessing data using Model files created by you or an administrator.

To begin accessing data with the driver:

1. Install the driver:
   a) After downloading the product, unzip the installer files to a temporary directory.
   b) From the installer directory, run the appropriate installer file to start the installer. The installer file takes the following form:

      `PROGRESS_DATADIRECT_ODBC_nn_WIN_xx_INSTALL.exe`

   c) Follow the prompts to complete installation.

   Note:

   The installer program supports multiple installation methods, including command-line and silent installations. For detailed instructions, refer to the *Progress DataDirect for ODBC Drivers Installation Guide*.

   ---

2. Before you can use your driver, you must set the PATH environment variable to include the path of the `jvm.dll` file of your Java™ Virtual Machine (JVM).
3. To configure the driver using the ODBC Administrator (GUI), start the ODBC Administrator from the Progress DataDirect program group. The GUI dialog allows you to configure the data source definitions in the Windows Registry or generate connection strings.

   Note: The Windows driver also supports using connection strings to connect to your service. For more information, see "Using a connection string."

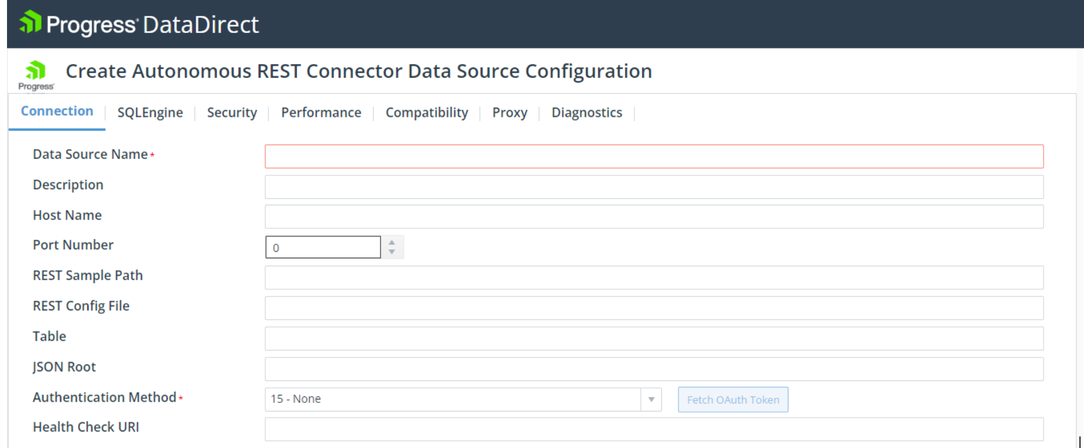
   ---

4. Select either the **User DSN**, **System DSN**, or **File DSN** tab to display a list of data sources.

- **User DSN**: If you installed a default DataDirect ODBC user data source as part of the installation, select the appropriate data source name and click **Configure** to display the driver Setup dialog box.

  If you are configuring a new user data source, click **Add** to display a list of installed drivers. Select your driver and click **Finish** to display the driver Setup dialog box.
- **System DSN**: To configure a new system data source, click **Add** to display a list of installed drivers. Select your driver and click **Finish** to display the driver Setup dialog box.

5. The **Connection** tab of the Configuration Manager opens.



Provide values for the following fields:

- **Data Source Name**: Type a string that identifies this data source configuration, such as `Projects`.
- **Description**: Type an optional long description of a data source name, such as `My Development Projects`.
- **Host Name**: Optionally, set this option to specify the host name portion of the HTTP endpoint to which you send requests. For example, a Jira endpoint would take the following form: `https://`*`mycompany`*`.atlassian.net`.

  Note: This value will be provided by the Model for data sources that have static host names.

- **Port Number**: Optionally, specify the TCP port of the server that is listening for REST API requests.
- **REST Config File**: Enter the name and location of your Model file. For example, `C:\path\to\myrest.rest`.

6. From the **Authentication Method** drop-down, select the authentication method used to connect to your data source. The properties related to the selected method are exposed. See Authentication on page 92 for more information on supported methods.
7. Provide values for the exposed authentication fields that apply to your connection. Note that some of the fields may not apply to your data source. Refer to the documentation for your REST service for details.
8. Set the values for any optional properties that you want to configure. For additional information on optional features and functionality, see the following resources:

   - Connection URL examples provides connection string examples that can be used to configure common functionality and features. You can modify and combine these examples to create a string that best suites your environment.

- Connection property descriptions provides a complete list of supported properties by functionality.

9. To test your configuration:
   a) Click **Test Connect**.
   b) In the **Test Query** field, specify a query that you would like to test.

   Note: The default query returns only a list of the tables defined in the Model file. This list is generated by the driver's internal SQL Engine. To authenticate to the REST service, specify a simple SELECT statement using of the table from the results.

   c) Click **Execute**.

   If successful, the driver will return results from a test query.

10. Click **Save** to save your data source.

    Note: As you provide values for properties, the Configuration Manager generates an ODBC connection string for use by your application. To use your string, click the Copy button ( ) and paste the string to a location that can be used by your application.

11. Begin accessing data with your applications, BI tools, database tools, and more. To help you get started, the following resources guide you through accessing data with some common tools:
    - Example Application: The example application allows you to test connect and execute SQL statements right out of the box.
    - Power BI: Power BI is a business intelligence software program that allows you to generate analytics and visualized representations of your data.
    - Tableau: Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data.
    - Microsoft Excel: Excel is a spreadsheet tool that allows you to connect, view tables, and execute SQL statements against your data.
    - Supported SQL statements and extensions on page 269: This section describes the syntax used for SQL statements supported by the driver. You can modify and use the provided examples for your application or tool.

This completes the deployment of the driver.

Installing and setting up the driver (Linux)

Before you begin: You will need a Model file before beginning this procedure. You can get one from an administrator or by creating your own (see Getting started creating a Model file on page 24). If you are connecting to a publicly available REST service, you can also use a pre-built Model file (see Getting started using prebuilt Model files (Windows) on page 20).

This section provides you with an overview of the steps required to install and set-up the driver. After completing this procedure, you will be able to begin accessing data with your application.

To begin accessing data with the driver:

1. Install the driver:
   a) After downloading the product, extract the contents of the product file.
   b) From the installer directory, run the installer's binary file to start the installer. The file for the installer program takes the following form:

      `PROGRESS_DATADIRECT_ODBC_nn_LINUX_xx_INSTALL.bin`

   c) Follow the prompts to complete installation.

   The installer program supports multiple installation methods, including command-line and silent installations. For detailed instructions, refer to the *Progress DataDirect for ODBC Drivers Installation Guide*.

2. Configure the environment variables:
   a) Check your permissions. You must log in as a user with full r/w/x permissions recursively on the entire product installation directory.
   b) Run one of the following product setup scripts from the installation directory to set variables: `odbc.sh` or `odbc.csh`. For Korn, Bourne, and equivalent shells, execute `odbc.sh`. For a C shell, execute `odbc.csh`. Executing the setup script:
      - Sets the ODBCINI environment variable to point to the path from the root directory to the system information file where your data source resides. For details, see "ODBCINI."
      - Sets the library path environment variable for your Linux operating system, `LD_LIBRARY_PATH`, to include the directory containing your JVM's `libjvm.so` file. For details, see "Library search path."

3. Configure the driver using one of the following methods:
   - **odbc.ini file**: You can begin using the driver immediately by editing the `odbc.ini` file in the installation directory with a text editor. The following demonstrates a data source definition with the minimal options.

     Basic Authentication

```
[ODBC Data Sources]
Autonomous REST Connector=DataDirect 8.0 Autonomous REST Connector


[Autonomous REST Connector]
Driver=ODBCHOME/lib/xxivautorest28.yy
...
AuthenticationMethod=0
...
RestConfigFile=C:/path/to/myrest.rest
...
UserName=jsmith
...
Password=secret
...
```

Note: The User and Password options are not required to be stored in the data source. They can also be sent separately by the application using the SQLConnect ODBC API. For SQLDriverConnect and SQLBrowseConnect, they will need to be specified in the data source or connection string.

OAuth 2.0 refresh token grant (Google Analytics)

```
[ODBC Data Sources]
Autonomous REST Connector=DataDirect 8.0 Autonomous REST Connector

[Autonomous REST Connector]
Driver=ODBCHOME/lib/xxivautorest28.yy
...
AuthenticationMethod=54
...
ClientID=123456789876-a1bc2de3fgh4ij567klmn8opqr9.apps.googleusercontent.com
...
ClientSecret=FaZBFRsGXTaR
...
RestConfigFile=C:/path/to/googleanalytics.rest
...
TokenURI=https://accounts.google.com/o/oauth2/token
...
```

See Configuration through the system information (odbc.ini) file on page 83 for more information.

Important: The client secret is a confidential value used to authenticate the application to the server. To prevent unauthorized access, this value must be securely maintained.

- **Connection string**: The driver also supports using connection strings for DSN (data source name), File DSN, or DSN-less connections. See Using a connection string on page 82, DSN-less connections, for more information. For examples, see Connection string examples on page 34.

Note: For most connections, specifying the minimum required connection options is sufficient to begin accessing data; however, you can provide values for optional connection options to use additional supported features and improve performance.

---

4. Set the values for any additional options that you want to configure. For additional information on optional features and functionality, see the following resources:

   - Connection string examples provides connection string examples that can be used to configure common functionality and features. You can modify and combine these examples to create a string that best suits your environment.

   Note: The options and values described in "Connection string examples" apply to all configuration methods.

   - Connection option descriptions provides a complete list of supported options by functionality.
   - Performance considerations describes connection options that affect performance, along with recommended settings.

5. Connect to your service and begin accessing data with your applications, BI tools, database tools, and more. To help you get started, the following resources guide you through accessing data with some common tools:

   - Example Application: The example application is a command-line tool that allows you to test connect, execute SQL statements, and practice using the ODBC API in environments that do not support GUIs.
   - Supported SQL statements and extensions: This section describes the syntax used for SQL statements supported by the driver. You can modify and use the provided examples for your application or tool.

This completes the deployment of the driver.

Connection string examples

ODBC provides a method for specifying connection information via a connection string and the SQLDriverConnect API. This section provides examples of connection strings configured to use common features and functionality. You can modify and/or combine these examples to create a connection string for your environment.

In addition to the connection strings for DSN-less connections demonstrated in this section, the driver supports DSN and File DSN connection strings. See "Using a connection string" for syntax and detailed information for supported connection string types.

Note: The options and values described in this section apply to all configuration methods.

Connection strings for the Autonomous REST Connector can take a few different forms. Typically, the composition of the string is dependent on whether you want to use a Model file, which requires configuring the Rest Config File (RESTConfigFile) option, or specify a single endpoint, which requires the Rest Sample Path (RestSamplePath) option. However, these options are not required in all scenarios, such as when the driver is executing dynamically created stored procedures or functions.

For sessions using a Model file (sessions that use multiple endpoints, POST requests, or other customizations supported by the Model file):

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;HostName=host_name;
       RestConfigFile=model_file;[attribute=value[;...]]
```

For sessions using the REST Sample Path option:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;RestSamplePath='sample_path';
       [attribute=value[;...]]
```

Note that all the examples in this section use the Model file.

- Basic authentication
- AWS credentials authentication
- Bearer token authentication
- Digest authentication
- HTTP header authentication
- URL parameter authentication
- Access token flow
- Authorization code grant
- Client credentials grant
- JWT bearer grant
- Password grant
- PKCE grant
- Refresh token grant

Basic authentication

This string includes the options used to connect using a Model file and basic authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=0;
       RestConfigFile=C:/path/to/myrest.rest;User=jsmith;Password=secret;
```

For more information on these options and values, see Basic authentication on page 93.

AWS credentials authentication

This string includes the properties used to connect to using a Model file with AWS credentials authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AccessKey=ABCDEFGHIJKL1EXAMPLE;
        AuthenticationMethod=43;Region=us-east-2;RestConfigFile=C:/path/to/myrest.rest;

        SecretKey=aBcdeFGhiJKLM/N1OPQRS/tUvWxyzYEXAMPLEKEY
```

For more information on these options and values, see AWS credentials authentication on page 94.

Bearer token authentication

This string includes the options used to connect using a Model file and bearer token authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=34;
       RestConfigFile=C:/path/to/myrest.rest;
       SecurityToken=C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx;
```

For more information on these options and values, see Bearer token authentication on page 95.

Digest authentication

This string includes the options used to connect using a Model file and digest authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=30;
       RestConfigFile=C:/path/to/myrest.rest;User=jsmith;Password=secret;
```

For more information on these options and values, see Digest authentication on page 96.

HTTP header authentication

This string includes the options used to connect using a Model file and HTTP header authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=25;
       RestConfigFile=C:/path/to/myrest.rest;SecurityToken=XaBARTsLZReM;
       [attribute=value[;...]]
```

For more information on these options and values, see HTTP header authentication on page 97.

URL parameter authentication

This string includes the options used to connect using a Model file and URL parameter authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=14;
       AuthParam=apikey;RestConfigFile=C:/path/to/myrest.rest;
       SecurityToken=XaBARTsLZReM;
```

For more information on these options and values, see URL parameter authentication on page 98.

Access token

This string includes the options used to connect using a Model file and access token flow authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;
       AccessToken=C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx;
       AuthenticationMethod=50;RestConfigFile=C:/path/to/yelp.rest;
```

For more information on these options and values, see Access token flow on page 99.

Authorization code grant

This string includes the options used to connect using a Model file and authorization code grant authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=51;
       OauthCode=xyz123abc;ClientID=abcdefghij1k2lmn3o4p5qr67s;
       RestConfigFile=C:/path/to/box.rest;TokenURI=https://api.box.com/oauth2/token;
```

For more information on these options and values, see Authorization code grant on page 101.

Client credentials grant

This string includes the options used to connect using a Model file and client credentials authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=53;
       ClientID=123456789876-a1bc2de3fgh4ij567klmn8opqr.apps.googleusercontent.com;
       ClientSecret=FaZBFRsGXTaR;RestConfigFile=C:/path/to/googleanalytics.rest;
       TokenURI=https://accounts.google.com/o/oauth2/token;
```

For more information on these options and values, see Client credentials grant on page 103.

JWT bearer grant

This string includes the options used to connect using a Model file and JWT bearer grant authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;
        RestConfigFile=C:/path/to/salesforce.rest;AuthenticationMethod=55;
        ClaimsIssuer='1a2b3c4d5e_6f7g8h9g';ClaimsSubject=jsmith@example.com;
        JWTCertStore=jwtcert.jks;JWTCertPassword=secret;JWTCertAlias=myAlias;
```

For more information on these options and values, see JWT bearer grant on page 108.

Password grant

This string includes the options used to connect using a Model file and password grant authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=52;
       RestConfigFile=C:/path/to/zendesk.rest;
       TokenURI=https://accounts.google.com/o/oauth2/token;User=jjones@example.com;
       Password=secretstuff;
```

For more information on these options and values, see Password grant on page 110.

PKCE grant

This string includes the options used to connect using a Model file and PKCE grant authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;RestConfigFile=C:/path/to/box.rest;
       AuthenticationMethod=56;ClientID='abcdefghik2lmn3o5qr67s';
       ClientSecret=FaZBFRsGXTaR;AuthURI=https://accounts.spotify.com/authorize;
       RedirectURI=https://localhost:8080;
       TokenURI='https://accounts.spotify.com/api/token';SQLEngineMode=2
```

For more information on these options and values, see PKCE grant on page 112.

Refresh token grant

This string includes the options used to connect using a Model file and refresh token grant authentication.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=54;
       ClientID=1234567898-a1bc2de3fgh4ij567klmn8opqr9stu.apps.googleusercontent.com
       Clientsecret=FaZBFRsGXTaR;RestConfigFile=C:/path/to/googleanalytics.rest;
       RefreshToken=1/abCD0F1GHijkLmNOPqrs_T2VWx3Y-Zabc45dE6FGh;
       TokenURI=https://accounts.google.com/o/oauth2/token;
```

For more information on these options and values, see Refresh token grant on page 115.

See also

Using a connection string on page 82

Version string information

The Autonomous REST Connector has a version string of the format:

```
XX.YY.ZZZZ(BAAAA, UBBBB, JDDDDDD)
```

The Driver Manager on Linux has a version string of the format:

```
XX.YY.ZZZZ(UBBBB)
```

The component for the Unicode conversion tables (ICU) has a version string of the format:

```
XX.YY.ZZZZ
```

where:

XX is the major version of the product.

YY is the minor version of the product.

ZZZZ is the build number of the driver or ICU component.

AAAA is the build number of the driver's bas component.

BBBB is the build number of the driver's utl component.

DDDDDD is the version of the Java components used by the driver.

For example:

```
08.01.0017 (B0254, U0180, J000109)
       |__|  |___|  |___|   |____|
      Driver  Bas    Utl     Java
```

On Windows, you can check the version string through the properties of the driver DLL. Right-click the driver DLL and select **Properties**. The Properties dialog box appears. On the Details tab, the **File Version** will be listed with the other file properties.

You can always check the version string of a driver on Windows by looking at the About tab of the driver's Setup dialog.

On Linux, you can check the version string by using the test loading tool shipped with the product. This tool, ivtestlib for 32-bit drivers and ddtestlib for 64-bit drivers, is located in `install_directory/bin`.

The syntax for the tool is:

```
ivtestlib shared_object
```

or

```
ddtestlib shared_object
```

For example, for the 32-bit driver on Linux:

```
ivtestlib ivsfrc28.so
```

returns:

```
08.01.0001 (B0002, U0001, J000003)
```

For example, for the Driver Manager on Linux:

```
ivtestlib libodbc.so
```

returns:

```
08.01.0001 (U0001)
```

For example, for the 64-bit Driver Manager on Linux:

```
ddtestlib libodbc.so
```

returns:

```
08.01.0001 (U0001)
```

For example, for the 32-bit ICU component on Linux:

```
ivtestlib libivicu28.so
08.01.0001
```

Note: The full path to the driver does not have to be specified for the test loading tool.

getFileVersionString function

Version string information can also be obtained programmatically through the function `getFileVersionString`. This function can be used when the application is not directly calling ODBC functions.

This function is defined as follows and is located in the driver's shared object:

```
const unsigned char* getFileVersionString();
```

This function is prototyped in the `qesqlext.h` file shipped with the product.

Data types

The following table lists native data types supported by the driver and how they are mapped to ODBC data types.

Table 1: Autonomous REST Connector Data Types

Autonomous REST Connector Data Type	ODBC Data Type
BigInt	SQL_BIGINT
Binary	SQL_BINARY
Bit	SQL_BIT
Boolean	SQL_BIT

Autonomous REST Connector Data Type	ODBC Data Type
Char	SQL_CHAR
Date	SQL_TYPE_DATE
Decimal	SQL_DECIMAL
Double	SQL_DOUBLE
Float	SQL_FLOAT
GUID	SQL_CHAR
Integer	SQL_INTEGER
JSON	SQL_VARCHAR
LongVarBinary	SQL_LONGVARBINARY
LongVarChar	SQL_LONGVARCHAR
NVarChar	SQL_UNICODE_VARCHAR
SmallInt	SQL_SMALLINT
Time	SQL_TYPE_TIME
Timestamp	SQL_TYPE_TIMESTAMP
TinyInt	SQL_TINYINT
VarBinary	SQL_VARBINARY
VarChar	SQL_VARCHAR

Driver specifications

This section describes the general functionality supported by the driver.

- **ODBC compliance**: The driver is compliant with the Open Database Connectivity (ODBC) 3.52 specification. The driver is ODBC core compliant and supports some Level 1 and Level 2 features.

  Refer to "ODBC API and scalar functions" in the *Progress DataDirect for ODBC Drivers Reference* for a list of supported API functions.

  The driver supports only the following Level 2 functions:

  - SQLColumnPrivileges
  - SQLDescribeParam
  - SQLForeignKeys

    - SQLPrimaryKeys
    - SQLProcedures
    - SQLTablePrivileges

- **Unicode support**: The driver is fully Unicode enabled. On Linux platforms, the driver supports both UTF-8 and UTF-16. On Windows platforms, the driver supports UCS-2/UTF-16 only.

  Refer to "Internationalization, localization, and Unicode" in the *Progress DataDirect for ODBC Drivers Reference* for details.

- **Isolation and lock levels**: Because transactions are not supported, the driver supports only the isolation level 0 (read uncommitted).

  Refer to "Locking and isolation levels" in the *Progress DataDirect for ODBC Drivers Reference* for details.

- **Connections and statements supported**: The driver supports multiple connections and multiple statements per connection.

Mapping objects to tables

Data mapping describes how elements are mapped between two distinct data models. To support SQL access to a REST service, the REST endpoint must be mapped to a relational schema. The driver automatically generates a relational view of your data when the Model file is loaded and/or any of the initial sampling of the data in the REST responses has been completed. When generating the relational view, the driver decompounds JSON, CSV, and XML documents returned by endpoints into parent-child tables. The following sections describe how the driver handles mapping of responses for their respective format.

Note:

- You can use the Qualify Normalized Names option to configure whether the names of relational tables normalized from array columns are derived directly from the column name or prefixed with parent object names. See the "Qualify Normalized Names" connection option description for details.
- After connecting, you can view the relational mapping of your model through the **ER Diagram** tab provided in the Autonomous REST Composer. See "Generating a Model file with the Autonomous REST Composer" for more information on using the Autonomous REST Composer.

The driver detects whether a response is in the CSV, JSON, or XML format. If a service supports multiple formats, the driver attempts to use JSON by default. JSON is a more compact format that typically performs better; however, there are some advantages to using the other formats. If you prefer to use a particular format in this scenario, you can specify it in the Accept header by using the `#headers` object in the table definition of the Model file. See "Request with custom HTTP headers" for more information.

See also

Normalizing JSON maps on page 43
Determining the primary key on page 48
Requests with custom HTTP headers on page 228
Qualify Normalized Names on page 174
Generating a Model file with the Autonomous REST Composer on page 70
Requests with custom HTTP headers on page 228

Mapping JSON responses

When generating the relational view, the driver decompounds JSON documents returned by endpoints into parent-child tables. The driver handles mapping in the following manner:

- Simple and nested objects are flattened and mapped to a parent table
- Arrays of objects and arrays of strings are mapped to related child tables
- If a JSON map is detected, it is normalized into a child table. See "Normalizing a JSON map" for a list of detectable map types and a description of normalizing JSON maps.

For example, the following JSON document contains nested objects in the `address` object, an array strings in the `vehicles` object, and an array of objects in the `pets` object.


```
{"resident_id":"ajx363",
 "name":"Sydney Smith",
 "address":{"street": "101 Main Street", "city": "Raleigh", "state": "NC"},
 "county":"Wake",
 "pets":[{"species":"dog","breed":"beagle","weight":"35"}],
 "vehicles":["car","boat","bicycle"]
},
{"resident_id":"tzn525",
 "name":"Cora Welch",
 "address":{"street":"191 First Street","city":"Chapel Hill","state":"NC"},
 "county":"Orange",
 "pets":[{"species":"pig","breed":"yorkshire","weight":"55"}]
 "vehicles": ["scooter","truck","bicycle"]
}
```


When generating the relational view, the driver decompounds native objects into separate, but related tables. The mapping of the sample JSON document produced one parent table and two child tables. In the parent table, simple objects, such as `name` and `county`, are flattened into corresponding relational columns. Nested objects are also flattened into relational columns; however, column names are formed by concatenating the name of the parent and nested objects, which are joined by an underscore character. For example, the `ADDRESS_STEET` column contains the values of the `street` object that is nested in the `address` object.

Note: When an endpoint features a top-level object that contains only arrays, instead of mapping an empty table, the driver omits the object's table from the relational view and promotes tables generated from the subordinate arrays to the top level. The driver's log records empty tables that are excluded from the relational view using the following message: `Empty table is not being persisted:` *`table_name`*.

Primary keys for parent tables are determined heuristically from the top-level fields in the document. For example, `resident_id`. However, if none of the fields are determined to be viable candidates, the driver generates a primary key column, `ROWID`. Be aware that, when a `ROWID` column is generated, the driver also flattens and maps objects into a single table. See "Determining the primary key" for more information. Note that you can designate a new primary key in a parent table using the Configuration Manager. For details, see "Customizing your Schema."

You can specify the name of the parent table using the Table (`Table`) connection option. If no value is specified, The name is derived from the endpoint from which the data was sampled. For example, for the endpoint `https://example.com/residents/2`, the table would be named `residents_2` by default.

Note: When using the REST Sample Path (`Sample`) connection option, the driver maps endpoints that consist of only a host name to the `URL_` parent table by default. You can specify a different table name using the Table option.

Note: If a naming conflict occurs, a suffix comprised of an underscore and numeral, starting at `1`, is appended to the relational name of an object. For example, if your table contains an object that would normally map to `POSITION`, your object would map column `POSITION_1` to avoid a conflict with the column used for composite keys.

The parent table for our example is named `RESIDENTS_2` and takes the following form:

Table 2: RESIDENTS_2

RESIDENT_ID (PK)	NAME	ADDRESS_STREET	ADDRESS_CITY	ADDRESS_STATE	COUNTY
ajx363	Sydney Smith	101 MAIN STREET	Raleigh	NC	Wake
tzn525	Cora Welch	191 FIRST STREET	Chapel Hill	NC	Orange

The data for the `pets` arrays of objects normalizes to `PETS` child tables. When discovered, the objects within an array are mapped to corresponding relational columns. For example, the `species` and `breed` array values from the `pets` array in the JSON sample, are mapped as columns to the following `PETS` table. A foreign key relationship to the parent table is provided by including the primary key of the parent in the child, in this case, `RESIDENT_ID`. The primary key of the child table is a composite key formed by the primary key of the parent table combined with the positional information contained in the `POSITION` column. If the array is nested multiple layers deep, additional positional columns for parent objects are mapped to insure that a unique key is used.

The child table for the `pets` array would take the following form:

Table 3: PETS

RESIDENTS_RESIDENT_ID (PK)	POSITION (PK)	SPECIES	BREED	WEIGHT
ajx363	0	dog	beagle	35
tzn525	0	pig	yorkshire	55

The information in the `vehicles` array of strings normalizes to the `VEHICLES` child table. The values of the array are mapped into a single relational column that corresponds to the name of the array. For example, the values for the `vehicles` array in the JSON sample, such as `car` and `boat`, map to the `VEHICLES` column in the `VEHICLES` table. To maintain a unique foreign key, the driver generates a `POSITION` common to differentiate from the duplicate primary keys derived from the parent table.

Table 4: VEHICLES

RESIDENTS_RESIDENT_ID (PK)	POSITION (PK)	VEHICLES
ajx363	0	car
ajx363	1	boat
ajx363	2	bicycle
tzn525	0	scooter
tzn525	1	truck
tzn525	2	bicycle

See also

Determining the primary key on page 48
Customizing your schema on page 78
Table on page 190
REST Sample Path on page 180

Normalizing JSON maps

A JSON map is a collection of key-value pairs that contain a set of unique keys. Typically, the keys are used for reference and, therefore, act as identifiers with a real world relationship, such as ID numbers, dates, or times. The driver will attempt to detect maps by recognizing patterns and formats in the keys when sampling an endpoint. For the driver to automatically recognize an object as a map, the object must have the following characteristics:

- The keys must me be one of the following types:
  - Numeric values
  - GUIDs
  - Dates formatted as YYYY-MM-DD
  - Times using the ISO format. The map may contain values with and without timezones in the same map.
  - Timestamps using the ISO format. The map may contain values with and without timezones in the same map.
- Every key in the object must be of the same type.

For example, the following JSON document contains a map that uses dates as keys:

```
{
 "1979-10-31":{"attendance":"12080","opponent":"Wildcats","result":"loss"},
 "1979-12-06":{"attendance":"34000","opponent":"Mustangs","result":"loss"},
 "1979-11-06":{"attendance":"8500","opponent":"Jets","result":"loss"},
}
```

The driver normalizes the key-value pairs in the JSON map to a child table. The fields in the map value are mapped into relational columns. For example, the `attendance` and `opponent` fields, are mapped to relational columns of the same name. The primary key is determined by the key portion of the key-value pair and maps to the `KEY` column by default.

When mapping the schema, the driver normalizes the key-value pairs in the JSON map to a child table. The fields in the map value are mapped into relational columns. For example, the `attendance` and `opponent` fields, are mapped to relational columns with corresponding names. Similarly, the key portion of the key-value pair is mapped to the `KEY` column by default. The primary key is determined by the type of the key column. For key values that are numeric values or GUIDS, the `KEY` column is used as the primary key. For key values that are date, time, or timestamp values, the driver generates primary keys in the `ROWID` column to avoid the possibility of non-unique primary keys (See “Determining the primary key” for more information).

The name of the parent table is derived from the endpoint from which the data was sampled. The parent table for our example is named `SEASON_RESULTS` and takes the following form:

Table 5: SEASON_RESULTS

ROWID (PK)	KEY	ATTENDANCE	OPPONENT	RESULT
unique_id	1979-10-31	12080	Wildcats	loss
unique_id	1979-12-06	34000	Mustangs	loss
unique_id	1979-11-06	8500	Jets	loss

Mapping XML responses

When generating the relational view, the driver decompounds XML documents returned by endpoints into parent-child tables. The driver handles mapping in the following manner:

- Simple and nested elements are flattened and mapped to a parent table
- Repeating elements that are nested in the elements of the same name are mapped to related child tables.
- Elements with attributes are mapped as columns with the attribute value returned as a column value.

For example, the following XML document contains nested elements in the `address` element and repeating nested elements in the `pets` and `vehicles` elements.

```
<residents>
    <resident>
        <resident_id>ajx363</resident_id>
        <name>Sydney Smith</name>
        <address>
            <street>101 Main Street</street>
            <city>Raleigh</city>
            <state>NC</state>
        </address>
        <county>Wake</county>
        <pets>
            <pet>
                <species>dog</species>
                <breed>beagle</breed>
                <weight unit="lbs">35</weight>
            </pet>
        </pets>
        <vehicles>
            <vehicle>car</vehicle>
            <vehicle>boat</vehicle>
```

```
                <vehicle>bicycle</vehicle>
            </vehicles>
        </resident>
        <resident>
            <resident_id>tzn525</resident_id>
            <name>Cora Welch</name>
            <address>
                <street>191 First Street</street>
                <city>Chapel Hill</city>
                <state>NC</state>
            </address>
            <county>Orange</county>
            <pets>
                <pet>
                    <species>pig</species>
                    <breed>yorkshire</breed>
                    <weight unit="lbs">55</weight>
                </pet>
            </pets>
            <vehicles>
                <vehicle>scooter</vehicle>
                <vehicle>truck</vehicle>
                <vehicle>bicycle</vehicle>
            </vehicles>
        </resident>
    </residents>
```

When generating the relational view, the driver decompounds native elements into separate, but related tables. The mapping of the sample XML document produced one parent table and two child tables. In the parent table, simple elements, such as `name` and `county`, are flattened into corresponding relational columns. Nested elements are also flattened into relational columns; however, column names are formed by concatenating the name of the parent and nested elements, which are joined by an underscore character. For example, the `ADDRESS_STREET` column contains the values of the `street` element that is nested in the `address` element.

Primary keys for parent tables are determined heuristically from the top-level fields in the document. For example, `resident_id`. However, if none of the fields are determined to be viable candidates, the driver generates a primary key column, `ROWID`. Be aware that, when a `ROWID` column is generated, the driver also flattens and maps objects into a single table. See "Determining the primary key" for more information. Note that you can designate a new primary key in a parent table using the Configuration Manager. For details, see "Customizing your Schema."

You can specify the name of the parent table using the Table (`Table`) connection option. If no value is specified, The name is derived from the endpoint from which the data was sampled. For example, for the endpoint `https://example.com/residents/2`, the table would be named `RESIDENTS_2` by default.

Note: When using the REST Sample Path (`Sample`) connection option, the driver maps endpoints that consist of only a host name to the `URL_` parent table by default. You can specify a different table name using the Table option.

Note: If a naming conflict occurs, a suffix comprised of an underscore and numeral, starting at `1`, is appended to the relational name of an element. For example, if your table contains an element that would normally map to `POSITION`, your element would map column `POSITION_1` to avoid a conflict with the column used for composite keys.

The parent table for our example is named `RESIDENTS` and takes the following form:

Table 6: RESIDENTS_2

RESIDENT_ID (PK)	NAME	ADDRESS_STREET	ADDRESS_CITY	ADDRESS_STATE	COUNTY
ajx363	Sydney Smith	101 MAIN STREET	Raleigh	NC	Wake
tzn525	Cora Welch	191 FIRST STREET	Chapel Hill	NC	Orange

The data for the repeating elements nested in the `pets` element normalizes to `PETS` child tables. When discovered, the repeating elements within a common parent element are mapped to corresponding relational columns. For example, the `species` and `breed` element values from the `pets` element in the XML sample, are mapped as columns to the following `PETS` table. A foreign key relationship to the parent table is provided by including the primary key of the parent in the child, in this case, `RESIDENT_ID`. The primary key of the child table is a composite key formed by the primary key of the parent table combined with the positional information contained in the `POSITION` column. If the array is nested multiple layers deep, additional positional columns for parent elements are mapped to insure that a unique key is used.

In addition, attributes contained in elements are mapped to the table as a discrete columns with the attribute value being mapped as the column value. For example, the `weight unit` maps to `WEIGHT_UNIT` with a value of `lbs`.

The child table for the `pets` array would take the following form:

Table 7: PETS

RESIDENTS_RESIDENT_ID (PK)	POSITION (PK)	SPECIES	BREED	WEIGHT_UNIT
ajx363	0	dog	beagle	lbs
tzn525	0	pig	yorkshire	lbs

The repeating elements nested in the `vehicles` element normalize to the `VEHICLES` child table. The values of the nested elements are mapped into a single relational column that corresponds to the name of the array. For example, the values for the elements nested in the `vehicles` element in the XML sample, such as `car` and `boat`, map to the `VEHICLES` column in the `VEHICLES` table. To maintain a unique foreign key, the driver generates a `POSITION` common to differentiate from the duplicate primary keys derived from the parent table.

Table 8: VEHICLES

RESIDENTS_RESIDENT_ID (PK)	POSITION (PK)	VEHICLES
ajx363	0	car
ajx363	1	boat
ajx363	2	bicycle
tzn525	0	scooter
tzn525	1	truck
tzn525	2	bicycle

See also

Determining the primary key on page 48
Customizing your schema on page 78
Table on page 190
REST Sample Path on page 180

Mapping CSV responses

When generating the relational view, the driver maps CSV (character-separated values) documents returned by endpoints into a single table. The driver handles mapping in the following manner:

- Values are mapped to a single table
- Arrays are mapped as strings in a single value
- Quoted, double-quoted, and escaped values are detected and processed by the driver. When mapping quoted values, the value has its quotation marks removed and is mapped as a literal value. For double-quoted and escaped values, the value has its external quotation marks and escapes removed and is mapped as a literal value, including any interior quotation marks.
- The following delimiters are supported: commas (`,`), tabs, pipes (`|`), colons (`:`), and semicolons (`;`).

For example, the following CSV document contains an escaped value in the `NAME` column and a quoted array in the `VEHICLES` column.

```
resident_id|name|street|city|state|county|pets|vehicles
ajx363|Sydney Smith|101 Main Street|Raleigh|NC|Wake|dog/beagle/35|"car,boat,bicycle"
tzn525|\"Cora" Welch\|191 First Street|Chapel
Hill|NC|Orange|pig/yorkshire/55|"scooter,truck,bicycle"
```

When generating the relational view, the driver maps native objects into a single table. Columns names are derived directly from the first entry of values in the document, for example `name` maps to `NAME`. Escaped values, such as `\"Cora" Welch\`, are mapped without their escapes, but retain capitalization, spacing, and punctuation. For example, `"Cora" Welch`. Similarly, quoted values have their external quotation marks removed, but are persisted as literal values in the relational map. For example, `"car,boat,bicycle"` is mapped as `car,boat,bicycle`.

Primary keys for parent tables are determined heuristically from the top-level fields in the document. For example, `resident_id`. However, if none of the fields are determined to be viable candidates, the driver generates a primary key column, `ROWID`. See "Determining the primary key" for more information. If necessary, you can designate a new primary key in a parent table using the Configuration Manager. See "Customizing your Schema" for details.

You can specify the name of the parent table using the Table property. If no value is specified, The name is derived from the endpoint from which the data was sampled. For example, for the endpoint `https://example.com/residents/2`, the table would be named `residents_2` by default.

Note: When using the Sample connection property, the driver maps endpoints that consist of only a host name to the `URL_` parent table by default. You can specify a different table name using the Table property.

Note: If a naming conflict occurs, a suffix comprised of an underscore and numeral, starting at `1`, is appended to the relational name of an object. For example, if your table contains an object that would normally map to `POSITION`, your object would map column `POSITION_1` to avoid a conflict with the column used for composite keys.

The table for our example is named `RESIDENTS_2` and takes the following form:

Table 9: RESIDENTS_2

RESIDENT_ID (PK)	NAME	STREET	CITY	STATE	COUNTY	PETS	VEHICLES
ajx363	Sydney Smith	101 Main street	Raleigh	NC	Wake	dog/beagle/35	car,boat,bicycle
tzn525	Cora Welch	191 First street	Chapel Hill	NC	Orange	pig/yorkshire/55	scooter,truck,bicycle

See also

Determining the primary key on page 48
Customizing your schema on page 78
Table on page 190

Determining the primary key

The primary key for parent tables are determined heuristically from the top-level fields in the document. When sampling, the driver attempts to find the first outermost simple column to designate as the primary key. Columns are then evaluated using the following rules to determine the most viable candidate:

- If sampling reveals a duplicate value, the column is not considered a good candidate
- If sampling reveals a null or empty value, the column is not considered a good candidate
- If sampling reveals certain statistical patterns in the content of the data, the column may be discarded as a candidate

- If sampling reveals a value of one of the following data types, the column is not considered a good candidate:
  - Array
  - Binary
  - Boolean
  - Date
  - Decimal
  - JSON
  - Time
  - Timestamp
  - String types greater than 32 characters (URLs greater than 128 characters)
- If sampling reveals a column name contains a common paging parameter, the column is not considered a good candidate
- If no top-level column is available, nested columns inside objects may be considered
- If the search fails to discover a viable candidate, the driver generates a primary key column named `ROWID`. When the driver generates a primary key column:
  - The driver flattens and maps the objects from the document into a single table. No child tables will be mapped from the document.
  - The primary key values are based on a composite of values in the row to create a unique value. Note that because the primary key is based on data, the primary key value might change between sessions if the data is modified.
  - The values of the `ROWID` column are only stored locally and persist for only the life of the session.

Note that this is just an overview of the rules employed by the driver. Additional and more subtle interactions occur when the driver encounters complex types or unusual data structures or values.

Designating a primary key

You can also designate a primary key other than the default using the Model file or the Autonomous REST Composer. For more information, see:

- "Primary key" to directly edit the Model file;
- "Customizing your schema" to use the Autonomous REST Composer.

In addition, when selecting your new primary key, you can review a list of potential primary key candidates by querying the `INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS` system table. For example:

```
SELECT * FROM INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS
```

The table contains a `PRIMARY_KEY_CANDIDATES` column, which includes a list of potential primary key candidates. The candidates are ranked in order by the driver starting with the best candidate. For more information on the `INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS` system table, see "Reviewing the status of your endpoints."

See also

Primary key on page 237
Customizing your schema on page 78
Reviewing the status of your endpoints on page 80

Troubleshooting

The *Progress DataDirect for ODBC Drivers Reference* provides information on troubleshooting problems should they occur.

Refer to the "Troubleshooting" section in the *Progress DataDirect for ODBC Drivers Reference* for details.

Contacting Technical Support

Progress DataDirect offers a variety of options to meet your support needs. Please visit our Web site for more details and for contact information:

https://www.progress.com/support

The Progress DataDirect Web site provides the latest support information through our global service network. The SupportLink program provides access to support contact details, tools, patches, and valuable information, including a list of FAQs for each product. In addition, you can search our Knowledgebase for technical bulletins and other information.

When you contact us for assistance, please provide the following information:

- Your number or the serial number that corresponds to the product for which you are seeking support, or a case number if you have been provided one for your issue. If you do not have a SupportLink contract, the SupportLink representative assisting you will connect you with our Sales team.
- Your name, phone number, email address, and organization. For a first-time call, you may be asked for full information, including location.
- The Progress DataDirect product and the version that you are using.
- The type and version of the operating system where you have installed your product.
- Any database, database version, third-party software, or other environment information required to understand the problem.
- A brief description of the problem, including, but not limited to, any error messages you have received, what steps you followed prior to the initial occurrence of the problem, any trace logs capturing the issue, and so on. Depending on the complexity of the problem, you may be asked to submit an example or reproducible application so that the issue can be re-created.
- A description of what you have attempted to resolve the issue. If you have researched your issue on Web search engines, our Knowledgebase, or have tested additional configurations, applications, or other vendor products, you will want to carefully note everything you have already attempted.
- A simple assessment of how the severity of the issue is impacting your organization.

January 2022, 8.0.1 Release of the Progress DataDirect Autonomous REST Connector for ODBC, Version 0001

2

Tutorials

The following sections guide you through using the driver to access your data with some common third-party applications. For information on installing your driver and setting the CLASSPATH, see "Installing and setting up the driver (Windows)," or "Installing and setting up the driver (Linux)."

For details, see the following topics:

- The Example application
- Power BI (Windows only)
- Tableau (Windows only)
- Microsoft Excel (Windows only)
- Connecting to REST APIs using the Composer user interface

The Example application

The driver installation includes an ODBC application called Example that can be used for:

- Testing any type of SQL statement
- Testing database connections
- Verifying your database environment

It can also be used to demonstrate ODBC function calls, including the following:

- SQLAllocHandle
- SQLBindCol
- SQLBindParameter
- SQLColAttribute
- SQLConnect
- SQLDescribeCol
- SQLDescribeParam
- SQLDisconnect
- SQLDriverConnect
- SQLExecDirect
- SQLFetch
- SQLFreeHandle
- SQLFreeStmt
- SQLGetDiagRec
- SQLGetInfo
- SQLNumResultCols
- SQLPrepare
- SQLSetEnvAttr
- SQLSetStmtAttr

The Example application can be built using the files located in the `\samples\examples` directory of the DataDirect for ODBC Drivers installation directory.

Note:

- For Windows, you can build the Windows app for ANSI and Unicode.
- For Linux, instructions for building the Example application are contained inside the file `example.mak`, which can be read with a text editor.

To use the Example application:

1. After you have configured the data source, navigate to the `instal_dir\samples\example` directory.
2. Open the application using one of the following methods:
   - Running the application executable or binary:
     - On Windows, double-click the `Example.exe` file.
     - On Linux, run the `example` application.
   - Executing a command-line argument. For example:
     - `example` *`connection_string`*
     - `example "DSN" "UID" "PWD"`
     - `example` *`connection_string`* `"`*`sql_command_1`*`" ["`*`sql_command_2`*`" ...]`

   Results: A command prompt opens.
3. Follow the prompts to enter your data source name, user name, and password. If successful, a SQL> prompt appears.
4. At the prompt, enter SQL statements to test your connection. For example:

   ```
   SELECT * FROM INFORMATION_SCHEMA.TABLES
   ```

The results of your query are displayed. If example is unable to connect, the appropriate error message is returned.

Power BI (Windows only)

After you have configured your data source, you can use the driver to access your data with Power BI. Power BI is a business intelligence software program that allows you to easily create reports and visualized representations of your data. By using the driver with Power BI, you can improve performance when retrieving data while leveraging the driver's relational mapping tools.

1. Navigate to the `\tools\Power BI` subdirectory of the Progress DataDirect installation directory; then, locate the installation batch file `install.bat`.
2. Run the `install.bat` file. The following operations are executed by running the `install.bat` file:
   - The Power BI connector file, `DataDirectAutoREST.pqx`, is copied to the following directory.

     `%USERPROFILE%\Documents\Power BI Desktop\Custom Connectors`
   - The following Windows registry entry is updated.

     `HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Power BI Desktop\TrustedCertificateThumbprints`
3. Open the Power BI desktop application.
4. From the **Get Data** window, navigate to **Other > Progress DataDirect Autonomous REST Connector**.
5. Click **Connect**. Then, from the **Progress DataDirect Autonomous REST Connector** window, provide the following information. Then, click **OK**.
   - **Data Source:** Enter a name for the data source. For example, `Autonomous REST Connector ODBC DSN`.
   - **SQL Statement:** If desired, provide a SQL command.
   - **Data Connectivity mode:**
     - Select **Import** to import data to Power BI.
     - Select **DirectQuery** to query live data. (For details, including limitations, refer to the Microsoft Power BI article Use DirectQuery in Power BI Desktop.)
6. Enter authentication information when prompted. Once connected, the **Navigator** window displays schema and table information.
7. Select and load tables. Then, prepare your Power BI dashboard as desired.

You have successfully accessed your data and are now ready to create reports with Power BI. For more information, refer to the Power BI product documentation at Power BI documentation.

Tableau (Windows only)

After you have configured your data source, you can use the driver to access your data with Tableau. Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data. By using the driver with Tableau, you can improve performance when retrieving data while leveraging the driver's relational mapping tools.

To use the driver to access data with Tableau:

1. Navigate to the `\tools\Tableau` subdirectory of the Progress DataDirect installation directory; then, locate the following Tableau data source file:

   `DataDirect Autonomous REST Connector.tdc`

2. Copy the Tableau data source file into the following directory:

   `C:\Users\`*`user_name`*`\Documents\My Tableau Repository\Datasources`

3. Open Tableau. If the **Connect** menu does not open by default, select **Data > New Data Source** or the Add New Data Source button to open the menu.
4. From the **Connect** menu, select **Other Databases (ODBC)**.
5. The **Other Databases (ODBC)** dialog appears. In the DSN field, select the data source you want to use from the drop-down menu. For example, **My DSN**. Then, click **Connect**. The Logon dialog appears pre-populated with the connection information you provided in your data source.
6. If required, type your user name and password; then, click **OK**. The Logon dialog closes. Then, click **Sign in** on the Other Databases (ODBC) dialog.
7. The **Data Source** window appears. By default, Tableau connects live, or directly, to your data. We recommend that you use the default settings to avoid extracting all of your data. However, if you prefer, you can import your data by selecting the **Extract** option at the top of the dialog.
8. In the Schema field, select the schema you want to use. The tables stored in this schema are now available for selection in the Table field.

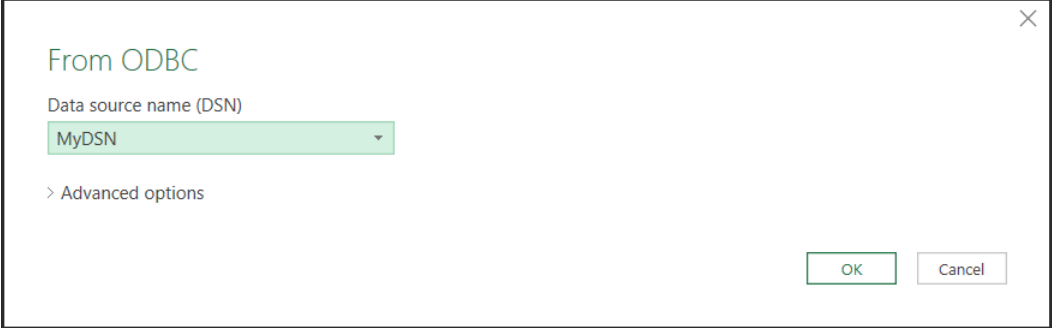
You have successfully accessed your data and are now ready to create reports with Tableau. For more information, refer to the Tableau product documentation at: http://www.tableau.com/support/help.

Microsoft Excel (Windows only)

After you have configured your data source, you can use the driver to access your data with Microsoft Excel from the Data Connection Wizard. Using the driver with Excel provides improved performance when retrieving data, while leveraging the driver's relational-mapping tools.

To use the driver to access data with Excel from the Data Connection Wizard:

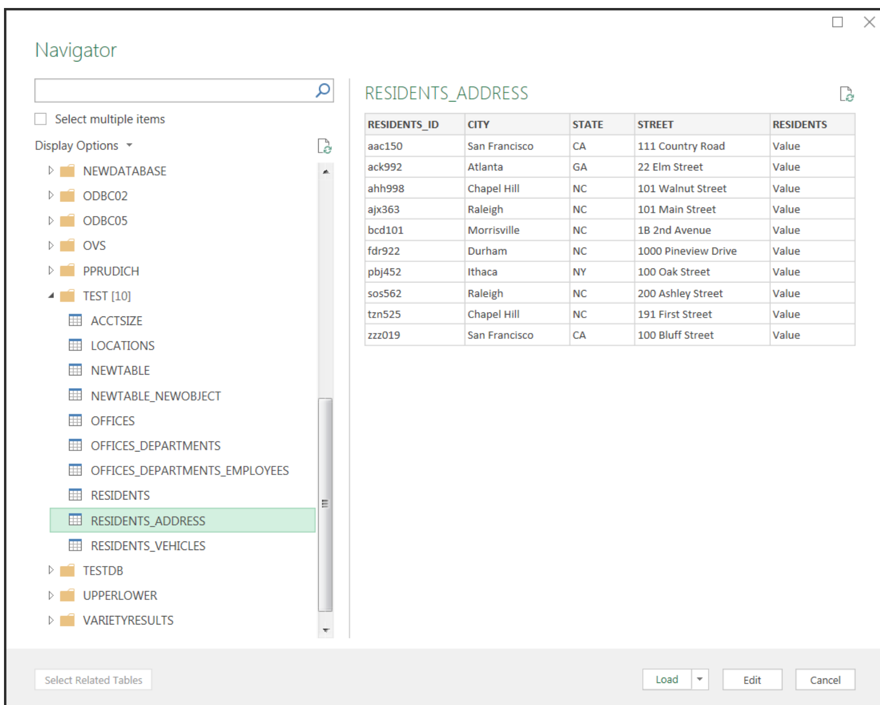
1. Open your workbook in Excel.
2. From the **Data** menu, select **Get Data**>**From Other Sources**>**From ODBC**.
3. The **From ODBC** dialog appears.



Select your data source from the Data Source Name (DSN) drop down; then, click **OK**.

4. You are prompted for logon credentials for your data source:
   - If your data source does not require logon credentials or if you prefer to specify your credentials using a connection string, select **Default or Custom** from the menu on the left. Optionally, specify your credential-related options using a connection string in the provided field. Click **Connect** to proceed.
   - If your data source uses Windows credentials, select **Windows** from the menu; then, provide your credentials. Optionally, specify a connection string with credential-related options in the provided field. Click **Connect** to proceed.
   - If your data source uses credentials stored on the database, select **Database**; then, provide your user name and password. Optionally, specify a connection string in the provided field. Click **Connect** to proceed.

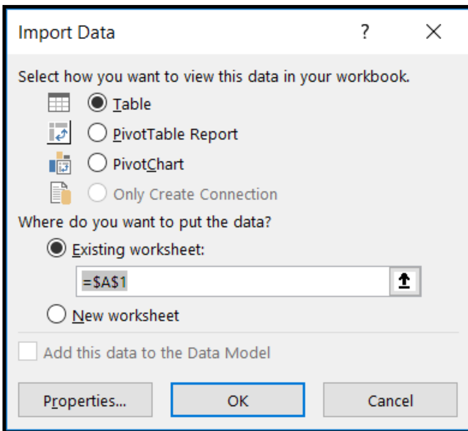
5. The **Navigator** window appears.



From the list, select the tables you want to access. A preview of your data will appear in the pane on the right. Optionally, click **Edit** to modify the results using the Query Editor. Refer to the Microsoft Excel product documentation for detailed information on using the Query Editor.

6. Load your data:
   - Click **Load** to import your data into your work sheet. Skip to the end.
   - Click **Load**>**Load To** to specify a location to import your data. Proceed to the next step.

7. The **Import Data** window appears.



Select the desired view and insertion point for the data. Click **OK**.

You have successfully accessed your data in Excel. For more information, refer to the Microsoft Excel product documentation at: https://support.office.com/.

Connecting to REST APIs using the Composer user interface

The Autonomous REST Connector uses sampling to normalize the JSON responses from REST APIs and make them available as relational tables. It empowers you to use the full capabilities of SQL, including joining data from different endpoints.

To get you started with the Autonomous REST Connector, in this tutorial, we will walk you through the process of connecting to a REST service from Tableau. This tutorial will give you an overview of how flexible Autonomous REST Connector is and how it works.

We will also use the Composer interface for the Autonomous REST Connector—which allows us to easily expose response data from multiple REST endpoints as tables that can be queried via standard SQL. The Composer interface is available as part of the Autonomous REST Connector for Windows platforms.

For the purposes of demonstration, we will use the REST API for SpaceX project as our data source for this tutorial. The SpaceX project is an open-source service that is free to use and does not require authentication credentials—which allows us to begin accessing data immediately.

This tutorial covers the following:

- Download and install the Autonomous REST Connector for Windows
- Create an ODBC Data Source
- Connect from Tableau
- Connect to multiple endpoints

Note: The purpose of this tutorial is to generally demonstrate accessing data from one or more endpoint(s) with the Autonomous REST Connector. However, if you want to connect to a publicly available REST service (such as Aha!, HubSpot, Jira, etc.), we recommend using the prebuilt Model files that are included with the connector. For details, refer to Getting started using prebuilt Model files (Windows).

Download and install the Autonomous REST Connector for Windows

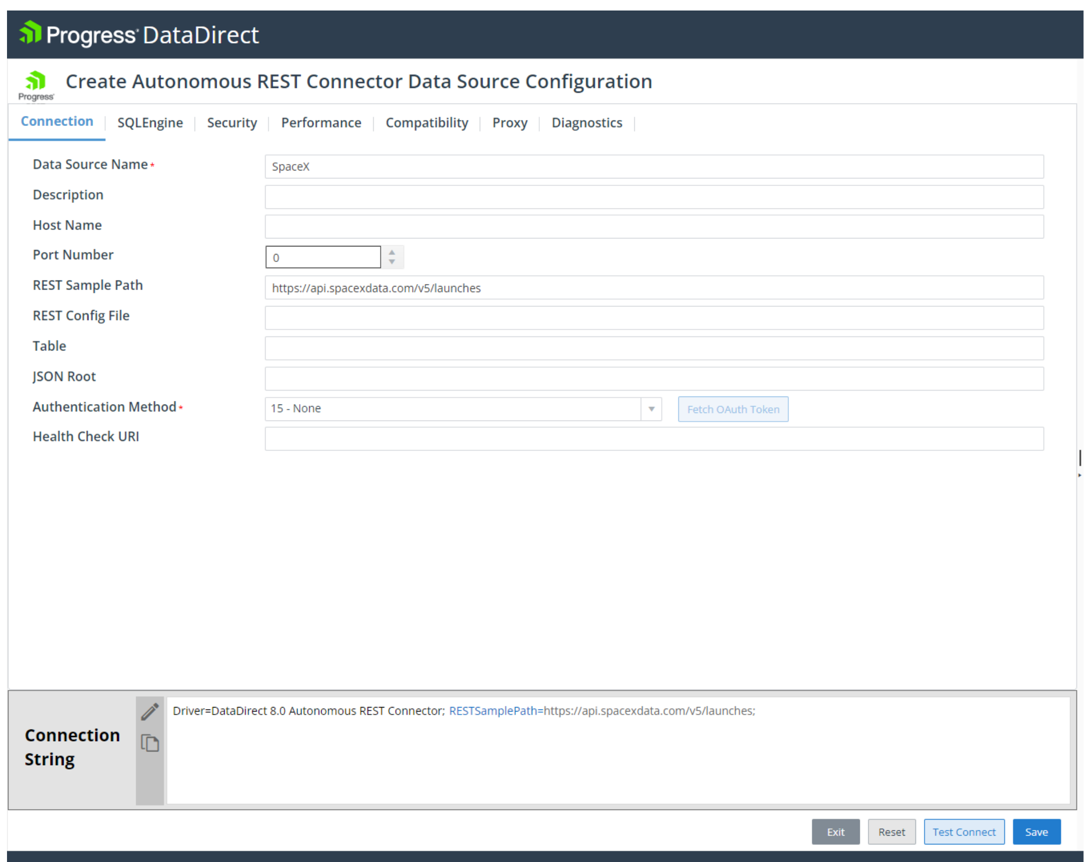
1. Navigate to the Autonomous REST Connector for ODBC page and select a Windows 32-bit or 64-bit connector to download. We recommend using the 64-bit driver if supported in your environment; however, Tableau supports either bitness.
2. When prompted, provide your details, such as name and email address; then, click **DOWNLOAD**.
3. Download the connector zip file and extract its contents into a temporary directory; then, double-click the installer program:
   - For 32-bit connectors: `PROGRESS_DATADIRECT_ODBC_8.0_WIN_32_INSTALL.exe`
   - For 64-bit connectors: `PROGRESS_DATADIRECT_ODBC_8.0_WIN_64_INSTALL.exe`
4. Follow the prompts to complete the installation.

Create an ODBC Data Source

1. Start the ODBC Administrator by selecting its icon from the Progress DataDirect for ODBC program group.
2. On the ODBC Administrator, select the **User DSN** tab, and then click **Add** to display a list of installed drivers. Select **DataDirect 8.0 Autonomous REST Connector** and click **Finish**.
3. The Configuration Manager appears in your default browser. On the **Connection** tab, specify values for the following fields:
   - **Data Source Name**: Specify a name for the data source you are creating. For example, `SpaceX`.
   - **REST Sample Path**: Specify the endpoint you want to connect to. For example, if you want to connect to the `launches` endpoint, specify:

     ```
     https://api.spacexdata.com/v5/launches
     ```

   - **Table**: Launches



4. Click **Test Connect** to connect to the SpaceX API. The Test Connect window opens. Optionally, execute queries in the **Test Query** field. For example:

   ```
   SELECT * FROM LAUNCHES
   ```

5. Click **Save**.

Connect from Tableau

1. Navigate to the `\tools\Tableau` subdirectory of the Progress DataDirect installation directory; then, locate the following Tableau data source file:

   ```
   DataDirect Autonomous REST Connector.tdc
   ```

2. Copy the Tableau data source file into the following directory:

   ```
   C:\Users\user_name\Documents\My Tableau Repository\Datasources
   ```

3. Open Tableau. If the Connect menu does not open by default, select **Data** > **New Data Source** or the **Add New Data Source** button to open the menu.
4. From the **Connect** menu, select **Other Databases (ODBC)**.
5. In the **DSN** field, select the data source you created in the Configuration Manager. Click **Connect**; then, **Sign In**. Your Tableau Book window opens.
6. In the **Schema** field, select **AUTOREST**. The tables stored in this schema are now available for selection in the **Table** field. For example, from the **Table** field, try dragging the **LAUNCHES** table into the canvas to view data restored in the JSON response from the REST API as relational data.

Result: You have successfully accessed your data and are now ready to create reports with Tableau. For more information, refer to the Tableau product documentation at: http://www.tableau.com/support/help.

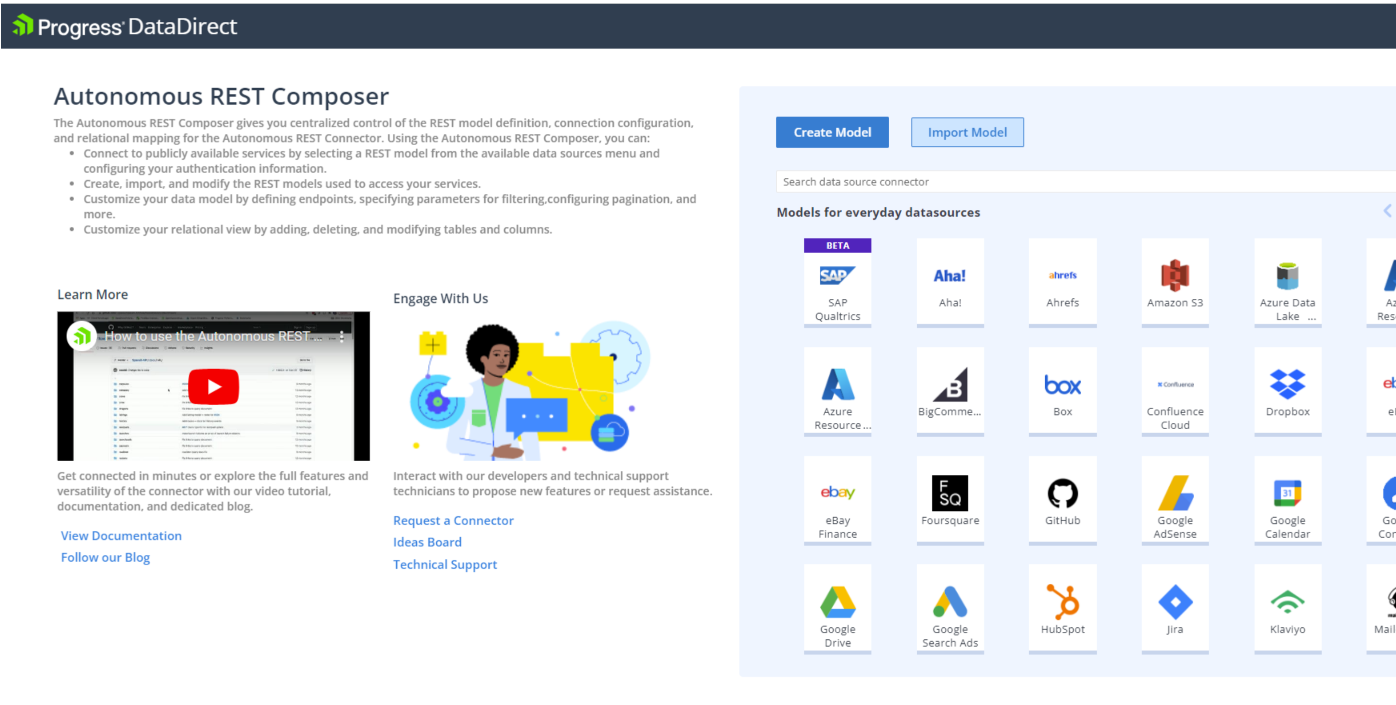
Connect to multiple endpoints

To connect to multiple endpoints using Autonomous REST Connector:

1. Open the Autonomous REST Composer by using one of the following methods:
   - Select the **Autonomous REST Composer (ODBC)** icon from your desktop or the Windows Start menu.
   - From a command line, navigate to the directory containing the `autorest.jar` file and execute the following command:

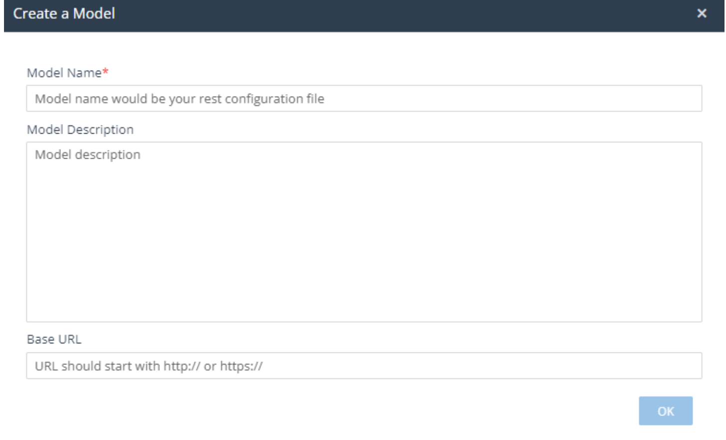
     ```
     java -jar autorest.jar --odbcdesign
     ```

     By default, the `autorest.jar` file is stored in the following directory: *install_dir*`\Progress\DataDirect\ODBC\java\lib\`.



2. Select **Create Model**. The **Create a Model** window appears.

Figure 5: The Create Model Window



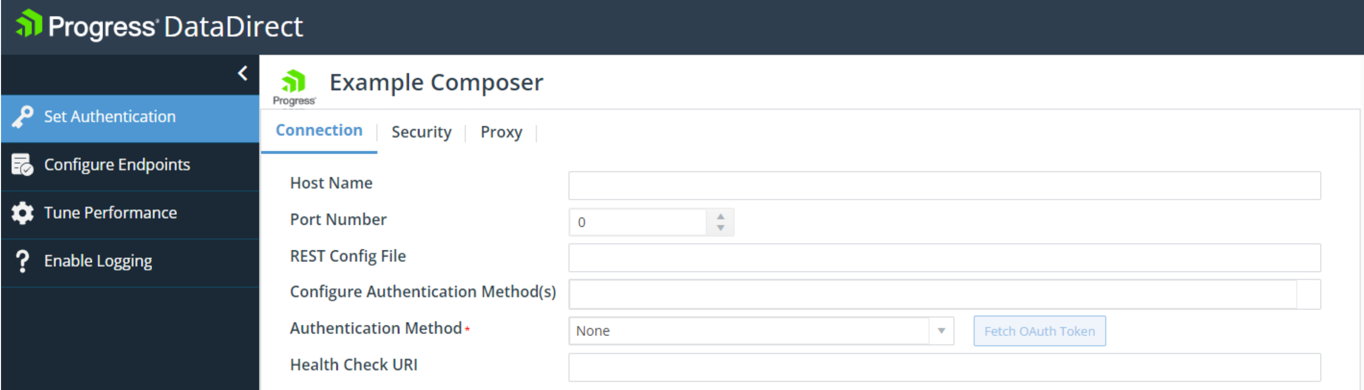
Complete the following fields to create a new project; then, click **OK**:

- **Model Name**: The name of your Model file to be created. For example: `Space X`.
- **Model Description**: An optional description of your Model. Note that this description will be stored in clear text in the Model file.
- **Base URL**: The host name portion of your REST endpoints. For example: `https://api.spacexdata.com`

  Note: The **Base URL** field is optional. You can also specify the base URL in the **Host Name** field on the **Connection** tab. Specifying a value in either location pre-populates the base URL in the Endpoints field(s) of the **Configure Endpoints** tab. Alternatively, you can also enter the entire URL for your endpoint into the Endpoints field.

The Composer opens to the **Connection** tab.

Figure 6: The Connection tab

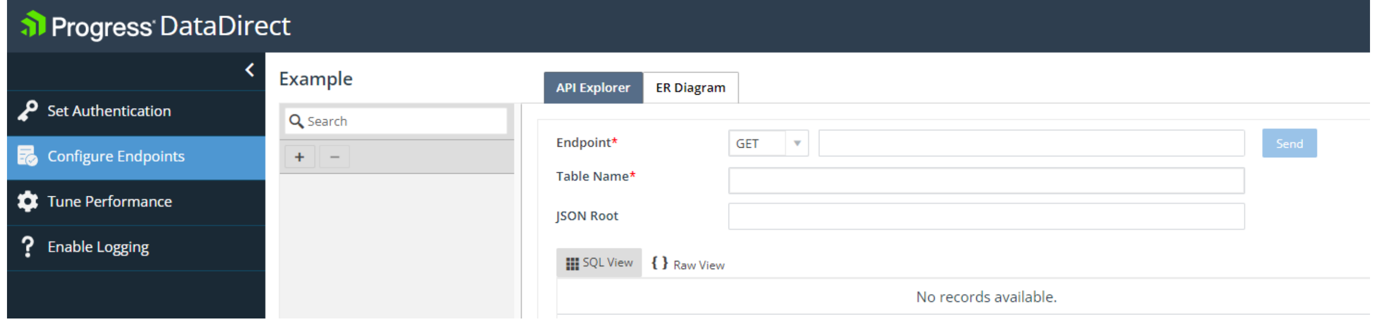


Note: Typically, the Connection tab is used to configure your authentication settings; however, for the data source used in this tutorial, no authentication settings are required.

---

3. Select the **Configure Endpoints** tab on the side menu.

Figure 7: The Configure Endpoints tab



Provide the minimum required information for an endpoint to which you want to issue requests:

- From the **Endpoint** drop-down menu, select the type of request to issue against your endpoint. For example: **GET**.
- In the **Endpoint** field, type your endpoint. Note that the value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`.

Note: If you provided a base URL using the **Base URL** field or Host Name option, specify only the path portion of your endpoint.

- In the **Table Name** field, type the name of the relational table to which you want the endpoint to map. For example: `LAUNCHES`.
- Optionally, in the JSON Root fields, type or edit the path to an embedded object that contains the results you want mapped to a dedicated relational table. You can use this option to expose only data starting at a nested field within a JSON response.

Note: During sampling, the driver specifies a default value for this field when it detects an embedded object that meets the criteria for a JSON root. Depending on the structure of your data model, you might want to add, edit, or remove values for this field.

For example, for the following JSON response, if you want to expose only the `resultsArray` data in a table named `RESULTS`, specify a **JSON Root** value of `/resultsArray` and a **Table Name** of `RESULTS`.

```
{
 "rootLevelField1": "exampleValue1",
 "rootLevelField2": "exampleValue2",
 "rootLevelField3": "exampleValue3",
 "resultsArray": [
     {
                        "id": 1,
                        "data": "exampleResults1"
     },
     {
                        "id": 2, "data":
                        "exampleResults2"
    }
  ]
}
```

Results: The data is displayed in a tabular format in the **SQL View**. The **Raw View** tab displays the JSON response body that was received from making the REST API call to the endpoint.

Note: Multiple tables might be mapped from a single endpoint. As a result of normalizing JSON data into a relational view, the driver exposes embedded array data as child tables. Conversely, parent tables are comprised of fields defined at the root level of the JSON response. A foreign key relationship to the parent table is provided by exposing the primary key of the parent in the child. You can use `SQL JOIN` syntax to rejoin the data of the parent and child tables into a single result set.

4. Click **Send**. The driver sends the REST request and generates a relational view of the data based on the response. To add additional endpoints, click in the **+** button the request list on the left.

For testing purposes, add the following endpoints:

Table 10: SpaceX Endpoints

Table Name	Endpoint
CAPSULES	https://api.spacexdata.com/v4/capsules
COMPANY	https://api.spacexdata.com/v4/company
CORES	https://api.spacexdata.com/v4/cores
CREW	https://api.spacexdata.com/v4/crew
DRAGONS	https://api.spacexdata.com/v4/dragons
HISTORY	https://api.spacexdata.com/v4/history
LANDPADS	https://api.spacexdata.com/v4/landpads
LAUNCHES	https://api.spacexdata.com/v5/launches
LAUNCHPADS	https://api.spacexdata.com/v4/launchpads
PAYLOADS	https://api.spacexdata.com/v4/payloads
ROADSTER	https://api.spacexdata.com/v4/roadster
ROCKETS	https://api.spacexdata.com/v4/rockets
SHIPS	https://api.spacexdata.com/v4/ships
STARLINK	https://api.spacexdata.com/v4/starlink

5. Click **Download** to generate and download your Model file. After downloading, move your Model file from the downloads folder to a directory that you have access to on your machine.
6. Open the ODBC Administrator and then double-click the data source you have created. The Configuration Manager appears.
7. On the **Connection** tab of the Configuration Manager:
   a. Clear the **REST Sample Path** field's value.
   b. In the **REST Config File** field, specify the fully qualified path and file name for the Model file you downloaded.
8. Click **Update**.
9. Open Tableau and reconnect to the data source.

Result: An updated list of tables is available based on the JSON responses from the endpoints defined in the Model file.

3

Configuring and connecting to data sources

After you install the driver, you configure data sources to connect to the database. Data sources store the Information that the driver needs to connect to a database. The ODBC specification describes three types of data sources: user data sources, system data sources (not a valid type on Linux), and file data sources. On Windows, user and system data sources are stored in the registry of the local computer. The difference is that only a specific user can access user data sources, whereas any user of the machine can access system data sources. On Windows and Linux, file data sources, which are simply text files, can be stored locally or on a network computer, and are accessible to other machines.

The data source contains connection options that allow you to tune the driver for specific performance. If you want to use a data source but need to change some of its values, you can either modify the data source or override its values at connection time through a connection string.

If you choose to use a connection string, you must use specific connection string attributes. See "Connection option descriptions" for an alphabetical list of driver connection string attributes and their initial default values.

For details, see the following topics:

- Environment settings
- Configuring the relational map
- Generating a Model file with the Autonomous REST Composer
- Configuring data sources with the Configuration Manager
- Generating connection strings with the Configuration Manager
- Using a connection string
- Additional configuration methods for Linux
- Testing connections and queries with the Configuration Manager

- Password Encryption Tool (UNIX/Linux only)
- Using a logon dialog box
- Using IP addresses
- Authentication
- TLS/SSL encryption
- Connecting through a proxy server
- Performance considerations

Environment settings

The first step in setting up and configuring the driver for use is to set environment settings and variables. The following procedures require that you have the appropriate permissions to modify your environment and to read, write, and execute various files. You must log in as a user with full r/w/x permissions recursively on the entire Progress DataDirect for ODBC installation directory.

Windows environment variables

Before you can use your driver, you must set the PATH environment variable to include the path of the `jvm.dll` file of your Java™ Virtual Machine (JVM).

Note: During installation, the Windows installer sets the PATH environment variable to include the path of the JVM.

Linux environment variables

The following topics guide you through setting the environment variables for Linux. You must set these environment variables before connecting with your driver.

Library search path

The library search path variable can be set by executing the appropriate shell script located in the ODBC home directory. From your login shell, determine which shell you are running by executing:

```
echo $SHELL
```

C shell login (and related shell) users must execute the following command before attempting to use ODBC-enabled applications:

```
source ./odbc.csh
```

Bourne shell login (and related shell) users must initialize their environment as follows:

```
. ./odbc.sh
```

Executing these scripts sets the library search path environment variable, `LD_LIBRARY_PATH`.

The library search path environment variable must be set so that the ODBC core components, Java components, and drivers can be located at the time of execution. After running the setup script, execute:

```
env
```

to verify that the *installation_directory*/lib directory has been added to your shared library path.

ODBCINI

The product installer places a default system information file, named odbc.ini, that contains data sources in the product installation directory. See "Configuration through the system information (odbc.ini) file" for an explanation of the odbc.ini file. The system administrator can choose to rename the file and/or move it to another location. In either case, the environment variable ODBCINI must be set to point to the fully qualified path name of the odbc.ini file.

For example, to point to the location of the file for an installation on /opt/odbc in the C shell, you would set this variable as follows:

```
setenv ODBCINI /opt/odbc/odbc.ini
```

In the Bourne or Korn shell, you would set it as:

```
ODBCINI=/opt/odbc/odbc.ini;export ODBCINI
```

As an alternative, you can choose to make the odbc.ini file a hidden file and not set the ODBCINI variable. In this case, you would need to rename the file to .odbc.ini (to make it a hidden file) and move it to the user's $HOME directory.

The driver searches for the location of the odbc.ini file as follows:

1. The driver checks the ODBCINI variable
2. The driver checks $HOME for .odbc.ini

If the driver does not locate the system information file, it returns an error.

See also

Configuration through the system information (odbc.ini) file on page 83

ODBCINST

The installer program places a default file, named odbcinst.ini, for use with DSN-less connections in the product installation directory. See "DSN-less connections" for an explanation of the odbcinst.ini file. The system administrator can choose to rename the file or move it to another location. In either case, the environment variable ODBCINST must be set to point to the fully qualified path name of the odbcinst.ini file.

For example, to point to the location of the file for an installation on /opt/odbc in the C shell, you would set this variable as follows:

```
setenv ODBCINST /opt/odbc/odbcinst.ini
```

In the Bourne or Korn shell, you would set it as:

```
ODBCINST=/opt/odbc/odbcinst.ini;export ODBCINST
```

As an alternative, you can choose to make the odbcinst.ini file a hidden file and not set the ODBCINST variable. In this case, you would need to rename the file to .odbcinst.ini (to make it a hidden file) and move it to the user's $HOME directory.

The driver searches for the location of the `odbcinst.ini` file as follows:

1. The driver checks the `ODBCINST` variable
2. The driver checks $HOME for `.odbcinst.ini`

If the driver does not locate the `odbcinst.ini` file, it returns an error.

See also

DSN-less connections on page 87

DD_INSTALLDIR

This variable provides the driver with the location of the product installation directory so that it can access support files. `DD_INSTALLDIR` must be set to point to the fully qualified path name of the installation directory.

For example, to point to the location of the directory for an installation on `/opt/odbc` in the C shell, you would set this variable as follows:

```
setenv DD_INSTALLDIR /opt/odbc
```

In the Bourne or Korn shell, you would set it as:

```
DD_INSTALLDIR=/opt/odbc;export DD_INSTALLDIR
```

The driver searches for the location of the installation directory as follows:

1. The driver checks the `DD_INSTALLDIR` variable
2. The driver checks the `odbc.ini` or the `odbcinst.ini` files for the InstallDir keyword (see "Configuration through the system information (`odbc.ini`) file" for a description of the InstallDir keyword)

If the driver does not locate the installation directory, it returns an error.

The next step is to test load the driver.

See also

Configuration through the system information (odbc.ini) file on page 83

The test loading tool

The second step in preparing to use a driver is to test load it.

The ivtestlib (32-bit driver) and ddtestlib (64-bit driver) test loading tools are provided to test load drivers and help diagnose configuration problems in the Linux environment, such as environment variables not correctly set or missing database client components. This tool is installed in the `/bin` subdirectory in the product installation directory. It attempts to load a specified ODBC driver and prints out all available error information if the load fails.

For example, if the driver is installed in `/opt/odbc/lib`, the following command attempts to load the 32-bit driver on Linux, where *xx* represents the version number of the driver:

```
ivtestlib /opt/odbc/lib/ivautorestxx.so
```

Note: The full path to the driver does not have to be specified for the tool.

If the load is successful, the tool returns a success message along with the version string of the driver. If the driver cannot be loaded, the tool returns an error message explaining why.

The next step is to configure a data source through the system information file.

UTF-16 applications on Linux

Because the DataDirect Driver Manager allows applications to use either UTF-8 or UTF-16 Unicode encoding, applications written in UTF-16 for Windows platforms can also be used on Linux platforms.

The Driver Manager assumes a default of UTF-8 applications; therefore, two things must occur for it to determine that the application is UTF-16:

- The definition of SQLWCHAR in the ODBC header files must be switched from "char *" to "short *". To do this, the application uses #define SQLWCHARSHORT.
- The application must set the encoding for the environment or connection using one of the following attributes. If your application passes UTF-8 encoded strings to some connections and UTF-16 encoded strings to other connections in the same environment, encoding should be set for the connection only; otherwise, either method can be used.
  - To configure the encoding for the environment, set the ODBC environment attribute SQL_ATTR_APP_UNICODE_TYPE to a value of `SQL_DD_CP_UTF16`, for example:

    ```
    rc = SQLSetEnvAttr(*henv, SQL_ATTR_APP_UNICODE_TYPE,
    (SQLPOINTER)SQL_DD_CP_UTF16, SQL_IS_INTEGER);
    ```

  - To configure the encoding for the connection only, set the ODBC connection attribute SQL_ATTR_APP_UNICODE_TYPE to a value of SQL_DD_CP_UTF16. For example:

    ```
    rc = SQLSetConnectAttr(hdbc, SQL_ATTR_APP_UNICODE_TYPE, SQL_DD_CP_UTF16,
    SQL_IS_INTEGER);
    ```

Configuring the relational map

The Autonomous REST Connector maps JSON responses to a relational model, exposing your REST data to relational, SQL-based applications. To map the responses to the relational model, the driver employs a Model file that defines endpoints and table mapping. The Model file is also capable of configuring a number of driver behaviors, such as paging, custom authentication, and HTTP response code processing. After creating a model file, you specify the location of the file using the REST Config File (RESTConfigFile) option to sample endpoints and begin accessing data.

The Model file is a simple text file that uses the *`file_name`*`.rest` naming convention. You can configure the file using either or both of the following methods:

- **The Autonomous REST Composer**: Using the included REST management tool, the Autonomous REST Composer, you can generate and edit REST model files. See "Generating a Model file with the Autonomous REST Composer."
- **Text editor**: Using a text editor, you can configure the relational map by populating the contents of the Model file with a list of comma-separated endpoints and requests using the formats described in "Model file syntax."

The following example demonstrates the basic format used in the Model file when mapping a table to the schema:

```
{
      "<table_name1>":"<endpoint1>",
      "<table_name2>":"<endpoint2>",
      "<table_name3>":"<endpoint3>"
      }
```

A sample Model file, named `example.rest`, is installed in the *install_dir*`\restfiles` directory. For additional examples, see "Sample Model file."

Note: The driver also offers a number of prebuilt Model files for publicly available data sources. See "Getting started using prebuilt Model files" for details.

See also

REST Config File on page 179
Generating a Model file with the Autonomous REST Composer on page 70
Model file syntax on page 199
Getting started using prebuilt Model files (Windows) on page 20

Generating a Model file with the Autonomous REST Composer

The Configuration Manager includes a REST management tool that allows you to generate and edit Model files. The primary purpose of the Model file is to define endpoint and table mapping, but it is also capable of configuring a number of driver behaviors, such as paging and pushdowns. After generating the Model file, you can share it among multiple installations of the driver.

To generate your Model file:

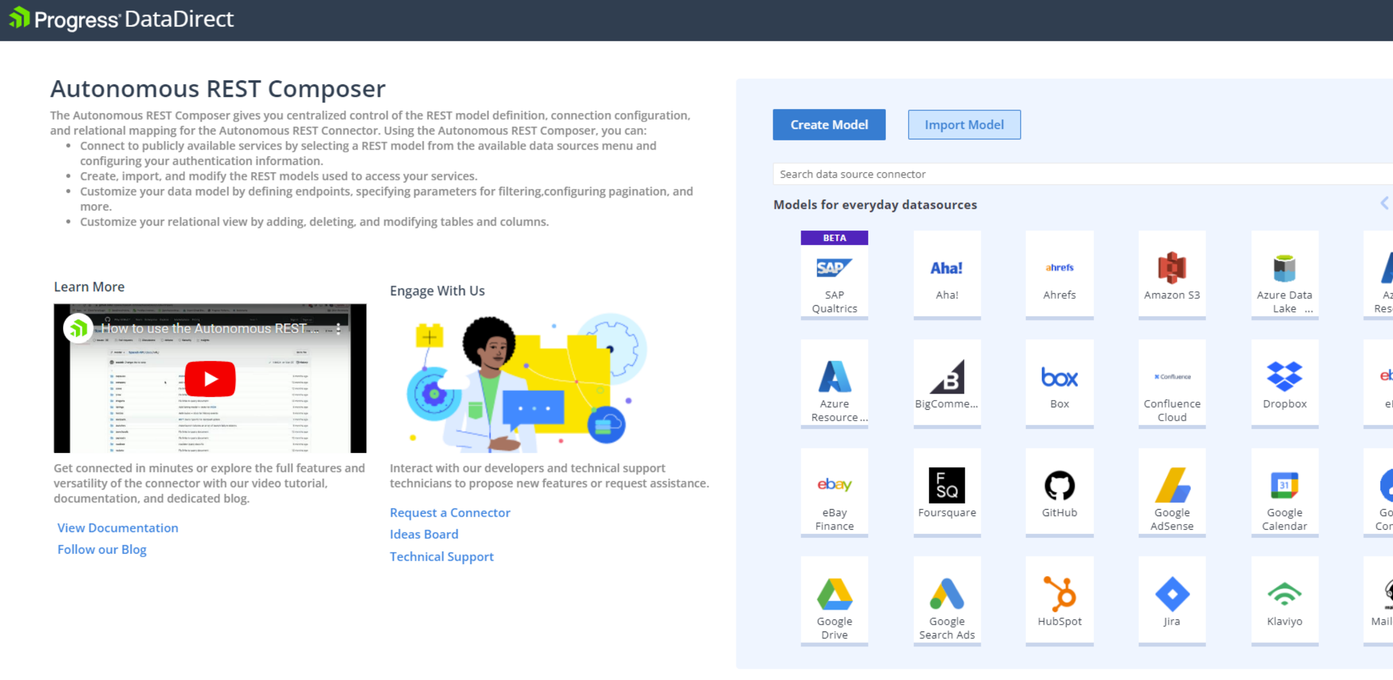
1. Open the Autonomous REST Composer by using one of the following methods:
   - Select the **Autonomous REST Composer (ODBC)** icon from your desktop or the Windows Start menu.
   - From a command line, navigate to the directory containing the `autorest.jar` file and execute the following command:

     ```
     java -jar autorest.jar --odbcdesign
     ```

     By default, the `autorest.jar` file is stored in the following directory: *install_dir*`\Progress\DataDirect\ODBC\java\lib\`.

The Autonomous REST Composer opens in your default web browser.

Figure 8: Hub window for Autonomous REST Connector



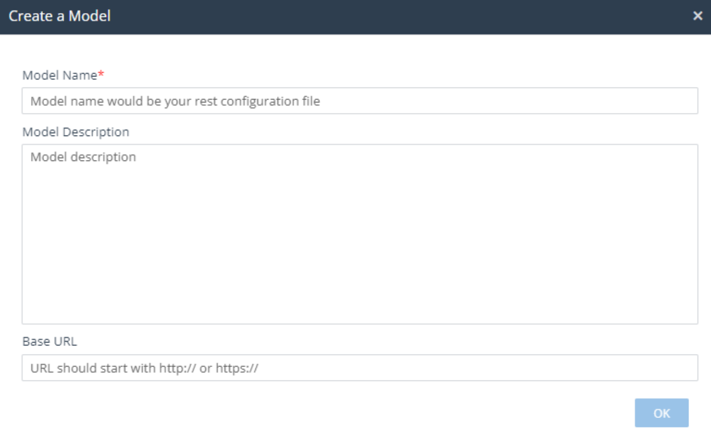
2. Select **Create a Model**.

Note: If you are accessing a publicly available data source, refer to the library of Model files for your data source. These Model files contain fully defined requests and pagination setting, allowing you to connect after providing your authentication credentials. See Getting started using prebuilt Model files (Windows) on page 20 for details.

---

3. The **Create Model** window opens.

Figure 9: The Create Model Window



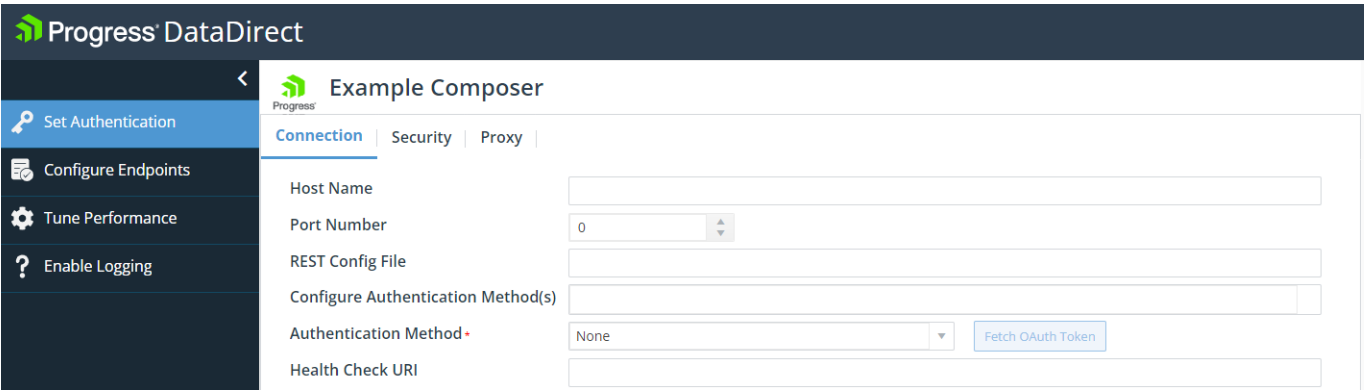
Complete the following fields to create a new project; then, click **OK**:

- **Model Name**: The name of your Model file to be created.

- **Model Description**: An optional description of your Model. Note that this description will be stored in clear text in the Model file.
- **Base URL**: The host name portion of your REST endpoints.

The Configuration Manager opens to the Connection tab.

Figure 10: The Connection tab



4. Select the Authentication Method used by your endpoints; then, provide values for the appropriate fields:

- In the Configure Authentication Method(s) field, select the method(s) used by your service.
- In the Authentication Method drop-down, select the method you want to configure. The fields associated with the method you select are exposed.
- In the exposed fields, provide values for the applicable fields. Note that not all of the fields exposed are required for all services.

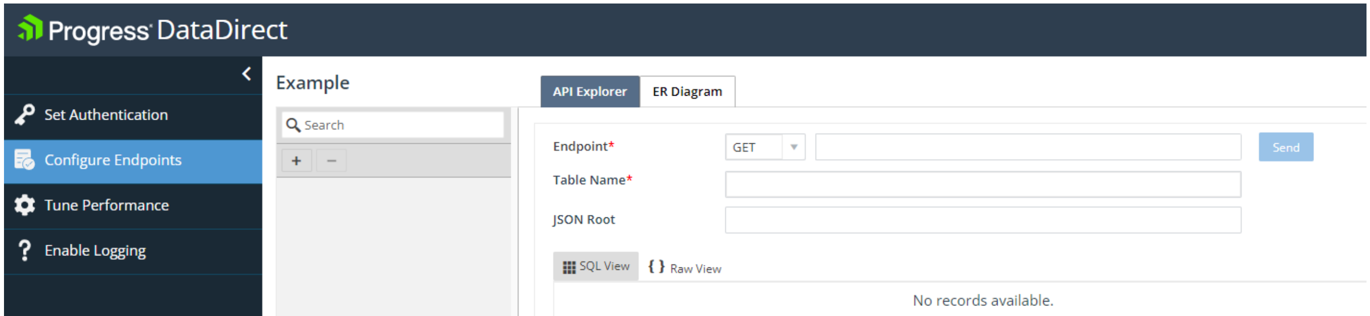
See Authentication on page 92 for a full description of these methods.

Note: Authentication properties specified through the Configuration Manager are not persisted in the Model file. To share authentication settings among all connections using the file, you must manually update the Model file. See OAuth 2.0 authentication on page 99 for details.

Note: Custom authentication properties are specified in the Model file. You can update the file manually or using the Configuration Manager. For details, see Custom authentication requests on page 206 and Configuring custom authentication with the Configuration Manager on page 78.

5. Select the **Configure Endpoints** tab on the side menu.

Figure 11: The Configure Endpoints tab

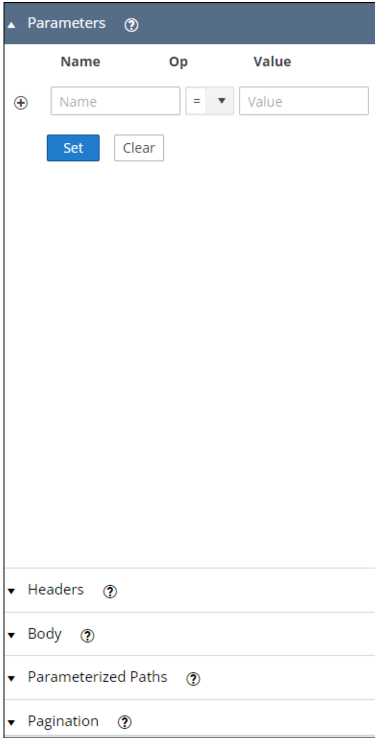


Provide the minimum required information for an endpoint to which you want to issue requests:

- From the **Endpoint** drop-down menu, select the type of request to issue against your endpoint.
- In the **Endpoint** field, type the path portion of your endpoint after the base URL. Note that the value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.
- In the **Table Name** field, type the name of the relational table to which you want the endpoint to map.

6. Optionally, further define your endpoint using the customization pane on the right. For example, specifying query parameters, parameterized paths, and POST request bodies. For detailed descriptions of defining different types of endpoints, see Sampling REST endpoints on page 74.

   Figure 12: Customization Pane

7. Click **Send**. The driver sends the REST request and generates a relational view of the data based on the response. To add additional endpoints, click **+** in the request pane on the left.
8. Optionally, in the **Pagination** section of the customization pane, select the paging method to be used for your Model; then, provide values for the applicable paging parameters. For a description of these parameters, see Paging on page 215.
9. Optionally, customize your relational schema, including modifying column names, data type mapping, and primary key designation. See Customizing your schema on page 78 for details.
10. Optionally, click **Test Connect** to test your model by executing SQL queries.
11. Optionally, you can review the status of your endpoints by querying the `SYSTEM_SAMPLING_STATUS` system table. For example:

   ```
   SELECT * FROM INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS
   ```

   This will allow you to verify that your endpoints have been successfully sampled by the driver and diagnose any issues, should they occur. See Reviewing the status of your endpoints on page 80 for details.

12. Click **Download** to generate and download your Model file.
13. Move your Model file to a location to be used by the driver. When configuring your connection string or data source, you will also need to specify this location using the REST Config File (`Config`) connection option.

After creating your Model file, you are ready to configure and connect. You can edit your Model file later by opening it from the **Create New Project** window when starting the Configuration Manager.

Note: The Model file generated by the Configuration Manager supports most of the request types and functionality typically used to access a service. However, the driver supports additional features and functionality that are not currently available through the file generated by the Configuration Manager. If you need to configure features or functionality not supported through the Configuration Manager, you can manually edit your generated file using a text editor. See "Model file syntax" for details.

See also

URL-encoded values on page 264
Model file syntax on page 199

Sampling REST endpoints

The following sections guide you through sampling different kinds of requests with Autonomous REST Composer:

- GET requests with unparameterized paths
- POST requests
- GET Requests with parameterized paths
- GET requests with query parameters
- GET requests with HTTP headers
- Requests for embedded objects

GET requests with unparameterized paths

To define a GET request with an unparameterized path:

1. From the **Endpoint** drop-down menu, select **GET**.
2. In the **Endpoint** field, type the path portion of your endpoint after the base URL. Note that the value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.
3. In the **Table Name** field, type the name of the relational table to which you want the endpoint to map.
4. Click **Send** to sample and map the endpoint.

POST requests

To use POST requests, you must define the path and the body of the request in the Model file in the JSON format. The path contains the URL endpoint and the body used in requests, while the body defines documents and provides sample values. The driver uses these sample values to define which data type to be used when executing a POST request.

To define a POST request:

1. From the **Endpoint** drop-down menu, select **POST**.
2. In the **Endpoint** field, type the path portion of your endpoint after the base URL. Note that the value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.
3. In the **Table Name** field, type the name of the relational table to which you want the endpoint to map.
4. In the Body pane:
    a. From the drop down menu, select the response format type for the payload.
    b. Enter the body of a POST request that is used to define documents and provide sample values. For example:

    ```
    {
     "start_date":"2018-08-31",
     "end_date":"2018-09-01",
     "departments":"[engineering,marketing,sales]",
     "tags":"[blue,green,red]"
    }
    ```

5. Click **Send** to sample and map the endpoint.

GET Requests with parameterized paths

Parameterized requests are issued as GET requests, unless they are specified in a POST request entry.

To define a request with parameterized paths:

1. From the **Endpoint** drop-down menu, select **GET**.
2. In the **Endpoint** field, type the path portion of your endpoint after the base URL. Note that the value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.
3. In the **Table Name** field, type the name of the relational table to which you want the endpoint to map.
4. In the **Paramaeterized Paths** pane, enter your parmaterized paths using the following format:

    ```
    <base_path>/{<param_name>:<param_value>}/<optional_path>
    ```

    For example:

    ```
    /orders/get/{date:2020-07-01,yyyy-MM-dd}/all
    ```

5. Click **+** to add additional parameterized paths for your endpoint.
6. Click **Send** to sample and map the endpoint.

GET requests with query parameters

Requests with query parameters are issued as GET requests, unless they are specified in a POST request entry.

To define a request with query parameters:

1. From the **Endpoint** drop-down menu, select **GET**.
2. In the **Endpoint** field, type the path portion of your endpoint after the base URL. Note that the value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.
3. In the **Table Name** field, type the name of the relational table to which you want the endpoint to map.
4. In the **Parameters** pane, enter your arguments used for filtering the request:
   - **Name**: The argument parameter component of the *`parameter=value`* pair used for filtering the request. For example, `interval`.
   - **Op**: The operator component of the argument.
   - **Value**: The value argument parameter used for filtering the request. This value is the default parameter value when issuing a query. You can override the default value by providing a parameter as a filtering condition in the Where clause of a Select statement.
5. Click the **+** button to add additional arguments.
6. Click **Send** to sample and map the endpoint.

GET requests with HTTP headers

Some endpoints employ custom HTTP headers to filter data returned by a GET request. This type of filtering is typically used to create multiple unique reports/tables from the same endpoint.

To define a request with HTTP headers:

1. From the **Endpoint** drop-down menu, select **GET**.
2. In the **Endpoint** field, type the path portion of your endpoint after the base URL. Note that the value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.
3. In the **Table Name** field, type the name of the relational table to which you want the endpoint to map.
4. In the **Headers** pane, provide values for the following fields:
   - **Name**: The HTTP header component of the *`header=value`* pair used for filtering the request. For example, `X-Subway-Payment`.
   - **Value**: The value argument for the HTTP header used for filtering the request or, if overriding the default Accept header, the value of the Accept header for the endpoint. For example, `token`.
5. Click **Send** to sample and map the endpoint.

Requests for embedded objects

Some endpoints store data that you want to access in embedded objects. During sampling, the driver attempts to detect these objects and map them to a dedicated table. However, in some scenarios, you might need to manually add or remove these embedded objects or edit their associated table names. The following steps guide you through manually specifying embedded objects for which you want to return results.

To return results for embedded objects:

1. From the **Endpoint** drop-down menu, select your request type.
2. In the **Endpoint** field, type the path portion of your endpoint after the base URL or select an existing endpoint from the pane on the left. Note that the endpoint value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.
3. In the **Table Name** field, provide or modify the name of the relational table to which you want the endpoint to map.
4. In the **JSON Root** field, provide or modify the simple path to the imbedded object containing the results you want mapped to a relational table. For nested elements, separate the element names with forward slashes (/).
5. Click **Send** to sample and map the endpoint.

See also

URL-encoded values on page 264

Parameter variables

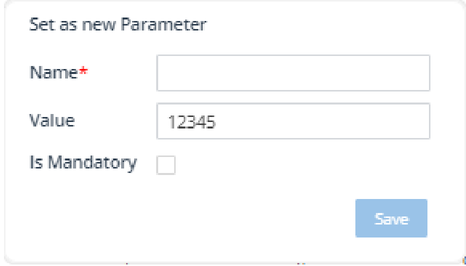
When creating or editing a REST model file, the parameter values you define in an endpoint might not apply to all users of the service. For example, using a parameter that filters results by your user ID or project might retrieve data relevant only to you. To facilitate sharing REST model files, the Composer allows you to designate parameter values in an endpoint as variables. Users with whom you share the file can then provide values for these variables that are replaced across their REST model file, allowing them to quickly customize the endpoints according to their use cases.

Replacing a parameter value with a variable

Before sharing a REST model file, you can replace parameter values that are user specific with variables. To replace a parameter variable with a variable in an endpoint:

1. Select the **Configure Endpoints** tab from the menu.
2. From the list of endpoints, expose and select the table that uses the parameter value you want to replace.
3. In the **Endpoints** field or **Parametrized Paths** pane, double-click on the parameter value in the path that you want to replace with a variable. The **Set as a new parameter** window opens.

Figure 13: Set as new parameter window in the Autonomous REST Composer



4. In the **Name** field, type the name of the variable that will replace your parameter value.
5. If the parameter value must be specified to return results, select the **Mandatory** check box. The mandatory box requires that a value is set for the parameter for the driver to query the associated table.
6. Click **Save** to apply your changes.

After you done configuring your REST model file, you can click **Download** to generate a copy of the REST Model file to share.

Setting parameter values for variables

After opening a shared REST model file in the Composer, you need to replace the variables with parameter values before querying data. To replace a parameter variable with a variable in an endpoint:

1. Select the **Configure Endpoints** tab from the menu.
2. From the list of endpoints, expose and select the table that uses a parameter value you want to define.
3. Open the **Params** pane to expose a list of parameters and value pairs that apply to the selected table. Note that a value must be specified for mandatory values.
4. In the **Value** field, provide
5. Click **Set** to apply your changes.

Follow this procedure for any table that uses parameter variables.

Customizing your schema

After sampling your endpoints, you can customize your schema from the default mapping, including modifying the column names, data type mapping, and primary key designation:

1. In the left pane, expand your sampled request to expose a list of your relational tables.
2. Under the table you want to edit, expand the list to expose the table columns.

Note: By right-clicking the table name, you can also edit the table name or delete the table.

3. Right-click on the column you want to edit; then, select **Modify Column Attributes**.

Note: By right-clicking the table name, you can also add or delete columns in the table.

Note: When designating a new primary key, you can query the `INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS` system table for a list of potential primary key candidates. See "Determining the primary key" for more information.

4. In the **Modify Column Attributes** window, edit the fields to your preferences.
5. Click **Save** to accept your changes.

See also

Determining the primary key on page 48

Configuring custom authentication with the Configuration Manager

Before you begin, determine the contents of the custom authentication entries to be used in your request. See Custom authentication requests on page 206 for detailed information on the supported syntax for custom authentication requests.

If your service does not support one of the standard authentication methods provided by the driver, you can define custom authentication requests to retrieve and exchange access tokens. You can configure custom authentication either by using the Autonomous REST Composer or by directly modifying the Model file. See Model file syntax on page 199 and Custom authentication requests on page 206 for information on configuring custom authentication directly in the Model file.

To configure custom authentication using the Autonomous REST Composer:

1. Select the **Set Authentication** tab on the side menu.
2. On the **Connection** tab, select **Custom** in the following fields:
   - Authentication Method(s)
   - Authentication Method
3. From the **?** menu in the upper right-hand corner, select **REST Configuration**. The contents of the Model file will display in the **REST Configuration** window.
4. In the REST Configuration window, enter your custom authentication entries into the Model file. Note that all entries should use valid JSON syntax and be comma separated. For example:

```
{
  "#options": {
    "authenticationMethod": {
      "default": "Custom",
      "choices": "Custom"
    }
  }
}
```

   Becomes the following when adding a simple token request flow:

```
{
  "#options": {
    "authenticationMethod": {
      "default": "Custom",
      "choices": "Custom"
    }
  },
  "#authentication" : [
    "api-key={CustomAuthParams[1]}",
    {
        "credentials": {
            "username": "{user}",
            "password": "{password}",
            "company": "{customAuthParams[2]}"
        }
    },
    "POST http://{serverName}/bearertoken",
    "HEADER Authentication=Bearer {/access-token}"
   ]
}
```

5. Close the REST Configuration window. The Composer automatically saves your changes to the Model file.
6. If applicable, provide values for the following fields:
   - **User**: Specify the user name used to connect to your REST service.
   - **Password**: Specify the password used to connect to your REST service.

- **Custom Authentication Parameters**: Specify the list of parameter values used by custom authentication requests that are defined in the Model file. This option allows you to configure parameter values used in custom authentication requests on a per connection basis, without having to hard code them, and securely pass them in a connection string or data source definition. See Custom Authentication Parameters on page 153 for details.

This completes configuring custom authentication using the Autonomous REST Composer.

Reviewing the status of your endpoints

After testing your connection, you can review the status of your specified endpoints to verify that they have been successfully sampled by the driver. Reviewing the status of your endpoints allows you to immediately identify any data omitted from the model and the potential causes. To review your endpoint status, query the `INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS` system table. For example:

```
SELECT * FROM INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS
```

The driver generates the `INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS` table after connecting to and sampling the endpoints specified in your Model. The table contains the following information:

- `ENDPOINT_NAME`: The name of the endpoint as defined in the Model file.
- `MESSAGE`: The HTTP response message.
- `SAMPLE_URI`: The endpoint that was used for the connection.
- `SUCCESS`: Indicates whether the endpoint was successfully sampled.
- `HTTP_STATUS`: The HTTP response status code.
- `PRIMARY_KEY_CANDIDATES`: A ranked list of potential primary key candidates that starts with the best-rated candidate.

Note that, in certain situations, you might need to set the Sampling Failure Tolerance option to `-1` (`SamplingFailureTolerance=-1`) to receive the status for all your endpoints. For example, if the number of endpoints in your model that are unreachable exceeds the sampling failure tolerance set by Sampling Failure Tolerance, the driver will fail the connection and stop attempting to sample the remaining endpoints specified in your Model file. In that scenario, you will need to reconfigure Sampling Failure Tolerance to audit your complete set of endpoints.

Configuring data sources with the Configuration Manager

The driver includes an enhanced setup dialog, Progress DataDirect Autonomous REST Connector Configuration Manager, that allows you to configure data sources, generate connection strings, test connections, and execute test queries. On Windows, data sources are stored in the Windows Registry. You can configure and modify data sources through the ODBC Administrator using a driver Setup dialog box, as described in this section.

Note: As you configure your data source, the Configuration Manager generates a corresponding connection string in the **Connection String** pane. To use your connection string, click the Copy button ( ) and paste the string to a location that can be used by your application. See "Generating connection strings with the Configuration Manager" for details.

Note: Connection string attributes can be used to override the default values of the data source if you want to change these values at connection time.

To configure and test a data source:

1. Open the Autonomous REST Connector Configuration Manager by selecting ODBC Administrator from the Progress DataDirect program group.
2. Open the Configuration Manager through the **User DSN** or **System DSN** tab.
   - **User DSN**: If you are configuring an existing user data source, select the data source name and click **Configure** to display the Configuration Manager in your browser.

     If you are configuring a new user data source, click **Add** to display a list of installed drivers. Select your driver and click **Finish** to display the Configuration Manager.
   - **System DSN**: If you are configuring an existing system data source, select the data source name and click **Configure** to display the Configuration Manager in your browser.

     If you are configuring a new system data source, click **Add** to display a list of installed drivers. Select the driver and click **Finish** to display the Configuration Manager.
   - **Note:** Configuring a new data source using the File DSN tab is not currently supported with the configuration manager.

   The Autonomous REST Connector Configuration Manager window opens.
3. From the Configuration Manager window, provide values of the connection options you want to configure in the corresponding fields. To view more options, select the tabs at the top of the page. See "Connection option descriptions" for descriptions of the supported options.

   Note: See "Connection string examples" for a list of required options used for different configurations. The options and settings described in that section apply to all methods of configuration.

4. At any point during the process, you can click **Test Connect** to attempt to connect to the service with your settings. In the Test Connection window:

   a) Provide values for any fields required by your service. Note that the information you enter in the logon dialog box during a test connect is not saved.

   b) Optionally, in the **Test Query** field, enter any SQL queries you want to execute during the test. For example:

   ```
   SELECT * FROM INFORMATION_SCHEMA.SYSTEM_TABLES
   ```

   c) Click **Execute**.

   If successful, the window displays a confirmation message and, if a query was specified, the results of the query.
5. Click **Save** to apply your values as the default when connecting with the data source.

See also

Generating connection strings with the Configuration Manager on page 82
Connection option descriptions on page 133

Generating connection strings with the Configuration Manager

Note: The Configuration Manager is currently supported only on Windows platforms.

The Progress DataDirect REST endpoints Configuration Manager supports generating connection strings that can be used with your application. To generate a connection string, create a data source as described in "Configuring data sources with the Configuration Manager." As you provide connection option values, the Configuration Manager generates a connection string in the **Connection String** pane that corresponds to the data source.

In addition to providing values for connection option fields, you can manually edit your string by clicking the Edit button (). Note that editing your connection string also changes the values for the data source.

After you are done configuring the connection options, click **Test Connect** to test your connection string. See "Testing connections and queries with the Configuration Manager" for more information.

To use your string, click the Copy button () and paste the string to a location that can be used by your application.

See also

Configuring data sources with the Configuration Manager on page 80
Testing connections and queries with the Configuration Manager on page 89

Using a connection string

If you want to use a connection string for connecting to a database, or if your application requires it, you must specify either a DSN (data source name), a File DSN, or a DSN-less connection in the string. The difference is whether you use the `DSN=`, `FILEDSN=`, or the `DRIVER=` keyword in the connection string, as described in the ODBC specification. A DSN or FILEDSN connection string tells the driver where to find the default connection information. Optionally, you may specify *attribute=value* pairs in the connection string to override the default values stored in the data source.

The DSN connection string has the form:

```
DSN=data_source_name[;attribute=value[;attribute=value]...]
```

The FILEDSN connection string has the form:

```
FILEDSN=filename.dsn[;attribute=value[;attribute=value]...]
```

The DSN-less connection string specifies a driver instead of a data source. All connection information must be entered in the connection string because the information is not stored in a data source.

The DSN-less connection string has the form:

```
DRIVER=[{]driver_name[}][;attribute=value[;attribute=value]...]
```

"Connection option descriptions" lists the long and short names for each attribute, as well as the initial default value when the driver is first installed. You can specify either long or short names in the connection string.

An example of a DSN connection string with overriding attribute values for the driver for Linux or Windows is:

```
DSN=AutoREST;USER=JohnQPublic;PWD=XYZZY
```

A FILEDSN connection string is similar except for the initial keyword:

```
FILEDSN=AutoREST.dsn;USER=JohnQPublic;PWD=XYZZY
```

A DSN-less connection string must provide all necessary connection information:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=Basic;
RESTConfigFile=C:\path\to\myrest.rest;USER=JohnQPublic;PWD=XYZZY
```

See also

Connection option descriptions on page 133
Connection string examples on page 34

Additional configuration methods for Linux

This section contains configuration methods that are specific to the Linux environment.

Configuration through the system information (odbc.ini) file

In the Linux environments, a system information file is used to store data source information. Setup installs a default version of this file, called `odbc.ini`, in the product installation directory. This is a plain text file that contains data source definitions.

To configure a data source manually, you edit the `odbc.ini` file with a text editor. The content of this file is divided into three sections.

Note: The driver and driver manager support ASCII and UTF-8 encoding in the `odbc.ini` file.

Refer to the "Character encoding in the odbc.ini and odbcinst.ini files" in *Progress DataDirect for ODBC Drivers Reference* for details.

At the beginning of the file is a section named `[ODBC Data Sources]` containing *`data_source_name=installed-driver`* pairs, for example:

```
AutoREST2=DataDirect 8.0 Autonomous REST Connector
```

The driver uses this section to match a data source to the appropriate installed driver.

The [ODBC Data Sources] section also includes data source definitions. The default odbc.ini contains a data source definition for the driver. Each data source definition begins with a data source name in square brackets, for example, [AutoREST2]. The data source definitions contain connection string *attribute=value* pairs with default values. You can modify these values as appropriate for your system. "Connection option descriptions" describes these attributes. See "Sample default odbc.ini file" for sample data sources.

The second section of the file is named [ODBC File DSN] and includes one keyword:

```
[ODBC File DSN]
DefaultDSNDir=
```

This keyword defines the path of the default location for file data sources (see "File data sources").

Note: This section is not included in the default odbc.ini file that is installed by the product installer. You must add this section manually.

The third section of the file is named [ODBC] and includes several keywords, for example:

```
[ODBC]
IANAAppCodePage=4
InstallDir=/opt/odbc
Trace=0
TraceFile=odbctrace.out
TraceDll=/opt/odbc/lib/ivtrc28.so
ODBCTraceMaxFileSize=102400
ODBCTraceMaxNumFiles=10
```

The IANAAppCodePage keyword defines the default value that the Linux driver uses if individual data sources have not specified a different value. See "IANAAppCodePage" in the "Connection option descriptions" for details.

For supported code page values, refer to "Code page values" in the *Progress DataDirect for ODBC Drivers Reference*.

The InstallDir keyword must be included in this section. The value of this keyword is the path to the installation directory under which the /lib and /locale directories are contained. The installation process automatically writes your installation directory to the default odbc.ini file.

For example, if you choose an installation location of /opt/odbc, then the following line is written to the [ODBC] section of the default odbc.ini:

```
InstallDir=/opt/odbc
```

Note: If you are using only DSN-less connections through an odbcinst.ini file and do not have an odbc.ini file, then you must provide [ODBC] section information in the [ODBC] section of the odbcinst.ini file. The driver and Driver Manager always check first in the [ODBC] section of an odbc.ini file. If no odbc.ini file exists or if the odbc.ini file does not contain an [ODBC] section, they check for an [ODBC] section in the odbcinst.ini file. See "DSN-less connections" for details.

ODBC tracing allows you to trace calls to the ODBC driver and create a log of the traces for troubleshooting purposes. The following keywords all control tracing: Trace, TraceFile, TraceDLL, ODBCTraceMaxFileSize, and ODBCTraceMaxNumFiles.

For a complete discussion of tracing, refer to "ODBC trace" in the *Progress DataDirect for ODBC Drivers Reference*.

See also

Connection option descriptions on page 133
Sample default odbc.ini file on page 85

File data sources on page 88
DSN-less connections on page 87

Sample default odbc.ini file

The following is a sample `odbc.ini` file that the installer program installs in the installation directory. All occurrences of ODBCHOME are replaced with your installation directory path during installation of the file. Values that you must supply are enclosed by angle brackets (< >). If you are using the installed `odbc.ini` file, you must supply the values and remove the angle brackets before that data source section will operate properly. Commented lines are denoted by the # symbol. This sample shows a 32-bit driver with the driver file name beginning with `iv`. A 64-bit driver file would be identical except that driver name would begin with `dd` and the list of data sources would include only the 64-bit drivers.

```
[ODBC Data Sources]
Autonomous REST Connector=DataDirect 8.0 Autonomous REST Connector

[Autonomous REST Connector]
Driver=ODBCHOME/lib/ivautorest28.so
Description=DataDirect 8.0 Autonomous REST Connector
AccessKey=
AccessToken=
ArrayNormalizationThreshold=12
AuthenticationMethod=15
AuthHeader=Authorization
AuthParam=
ClientCredentialsMode=0
ClientID=
ClientSecret=
CreateMap=3
CryptoProtocolVersion=TLSv1.2, TLSv1.1, TLSv1
CustomAuthParams=
DebugRecord=
DefaultQueryOptions=
EnableLoginPrompt=0
EncryptionMethod=1
ExtendedOptions=
FetchSize=100
HealthURI=
HostName=
HostNameInCertificate=
JSONRoot=
JVMArgs=-Xmx256m
JVMClasspath=
KeyPassword=
Keystore=
KeystorePassword=
LogConfigFile=
LoginTimeout=15
LogoffURI=
OauthCode=
Password=
PortNumber=
ProxyHost=
ProxyPassword=
ProxyPort=
ProxyUser=
ReadAhead=0
RedirectURI=
RefreshDirtyCache=
RefreshSchema=0
RefreshToken=
Region=
ReportCodepageConversionErrors=0
RestConfigFile=
RestSamplePath=
ResultMemorySize=
```

```
SamplingFailureTolerance=0
SchemaMap=
Scope=
SecretKey=
SecurityToken=
SQLEngineMode=2
StmtCallLimit=0
StmtCallLimitBehavior=1
Table=
TokenURI=
TransactionMode=0
Truststore=
TruststorePassword=
User=
UserAgent=
ValidateServerCertificate=1
WSFetchSize=2000
WSPoolSize=1
WSRetryCount=5
WSTimeout=120

[ODBC]
IANAAppCodePage=4
InstallDir=ODBCHOME
Trace=0
TraceFile=odbctrace.out
TraceDll=ODBCHOME/lib/ivtrc28.so
ODBCTraceMaxFileSize=102400
ODBCTraceMaxNumFiles=10

[ODBC File DSN]
DefaultDSNDir=
UseCursorLib=0
```

To modify or create data sources in the `odbc.ini` file, use the following procedures.

- **To modify a data source:**

  a) Using a text editor, open the `odbc.ini` file.

  b) Modify the default values for attributes in the data source definitions as necessary based on your system specifics.

  See "Connection option descriptions" for other specific attribute values.

  c) After making all modifications, save the `odbc.ini` file and close the text editor.

  Important: The "Connection option descriptions" section lists both the long and short names of the attribute. When entering attribute names into `odbc.ini`, you must use the long name of the attribute. The short name is not valid in the `odbc.ini` file.

- **To create a new data source:**

  a) Using a text editor, open the `odbc.ini` file.

  b) Copy an appropriate existing default data source definition and paste it to another location in the file.

  c) Change the data source name in the copied data source definition to a new name. The data source name is between square brackets at the beginning of the definition, for example, `[My Datasource]`.

  d) Modify the attributes in the new definition as necessary based on your system specifics.

  See "Connection option descriptions" for other specific attribute values.

e) In the [ODBC] section at the beginning of the file, add a new *data_source_name=installed-driver* pair containing the new data source name and the appropriate installed driver name.

f) After making all modifications, save the odbc.ini file and close the text editor.

Important: The "Connection option descriptions" section lists both the long and short name of the attribute. When entering attribute names into odbc.ini, you must use the long name of the attribute. The short name is not valid in the odbc.ini file.

See also

Connection option descriptions on page 133

DSN-less connections

Connections to a data source can be made via a connection string without referring to a data source name (DSN-less connections). This is done by specifying the "DRIVER=" keyword instead of the "DSN=" keyword in a connection string, as outlined in the ODBC specification. A file named odbcinst.ini must exist when the driver encounters DRIVER= in a connection string.

Setup installs a default version of this file in the product installation directory (see "ODBCINST" for details about relocating and renaming this file). This is a plain text file that contains default DSN-less connection information. You should not normally need to edit this file. The content of this file is divided into several sections.

Note: The driver and driver manager support ASCII and UTF-8 encoding in the odbcinst.ini file.

Refer to the "Character encoding in the odbc.ini and odbcinst.ini files" in *Progress DataDirect for ODBC Drivers Reference* for details.

At the beginning of the file is a section named [ODBC Drivers] that lists installed drivers, for example,

```
DataDirect 8.0 Autonomous REST Connector=Installed
```

This section also includes additional information for each driver.

The final section of the file is named [ODBC]. The [ODBC] section in the odbcinst.ini file fulfills the same purpose in DSN-less connections as the [ODBC] section in the odbc.ini file does for data source connections. See "Configuration through the system information (odbc.ini) file" for a description of the other keywords this section.

Note: The odbcinst.ini file and the odbc.ini file include an [ODBC] section. If the information in these two sections is not the same, the values in the odbc.ini [ODBC] section override those of the odbcinst.ini [ODBC] section.

See also

ODBCINST on page 67

Configuration through the system information (odbc.ini) file on page 83

Sample odbcinst.ini file

The following is a sample `odbcinst.ini`. All occurrences of ODBCHOME are replaced with your installation directory path during installation of the file. Commented lines are denoted by the # symbol. This sample shows a 32-bit driver with the driver file name beginning with `iv`; a 64-bit driver file would be identical except that driver names would begin with `dd`.

```
[ODBC Drivers]
DataDirect 8.0 Autonomous REST Connector=Installed

[DataDirect 8.0 Autonomous REST Connector]
Driver=ODBCHOME/lib/ivautorest28.so
JarFile=ODBCHOME/java/lib/autorest.jar
APILevel=0
ConnectFunctions=YYY
CPTimeout=60
DriverODBCVer=3.52
FileUsage=0
HelpRootDirectory=ODBCHOME/Help/AutoRESTHelp
SQLLevel=0
UsageCount=1

[ODBC]
#This section must contain values for DSN-less connections
#if no odbc.ini file exists. If an odbc.ini file exists,
#the values from that [ODBC] section are used.

InstallDir=ODBCHOME
Trace=0
TraceFile=odbctrace.out
TraceDll=ODBCHOME/lib/ivtrc28.so
ODBCTraceMaxFileSize=102400
ODBCTraceMaxNumFiles=10
```

File data sources

The Driver Manager on Linux supports file data sources. The advantage of a file data source is that it can be stored on a server and accessed by other machines, either Windows or Linux. See "Configuring and connecting to data sources" for a general description of ODBC data sources on both Windows and Linux.

A file data source is simply a text file that contains connection information. It can be created with a text editor. The file normally has an extension of `.dsn`.

For example, a file data source for the driver would be similar to the following:

```
[ODBC]
Driver=DataDirect 8.0 Autonomous REST Connector
...
AuthenticationMethod=0
...
Password=secret
...
RestConfigFile=C:/path/to/myrest.rest
...
User=jsmith
...
```

It must contain all basic connection information plus any optional attributes. Because it uses the `DRIVER=` keyword, an `odbcinst.ini` file containing the driver location must exist (see "DSN-less connections").

The file data source is accessed by specifying the `FILEDSN=` instead of the `DSN=` keyword in a connection string, as outlined in the ODBC specification. The complete path to the file data source can be specified in the syntax that is normal for the machine on which the file is located. For example, on Windows:

```
FILEDSN=C:\Program Files\Common Files\ODBC\DataSources\AutoREST2.dsn
```

or, on Linux:

```
FILEDSN=/home/users/john/filedsn/AutoREST2.dsn
```

If no path is specified for the file data source, the Driver Manager uses the DefaultDSNDir property, which is defined in the `[ODBC File DSN]` setting in the `odbc.ini` file to locate file data sources (see "Configuration through the system information (odbc.ini) file" for details). If the `[ODBC File DSN]` setting is not defined, the Driver Manager uses the InstallDir setting in the `[ODBC]` section of the `odbc.ini` file. The Driver Manager does not support the SQLReadFileDSN and SQLWriteFileDSN functions.

As with any connection string, you can specify attributes to override the default values in the data source:

```
FILEDSN=/home/users/john/filedsn/AutoREST2.dsn;User=jsmith;PWD=test01
```

See also

Configuring and connecting to data sources on page 65
DSN-less connections on page 87
Configuration through the system information (odbc.ini) file on page 83

Testing connections and queries with the Configuration Manager

Note: The Configuration Manager is currently supported only on Windows platforms.

You can quickly test data sources, connection strings and queries using Progress DataDirect Autonomous REST Connector Configuration Manager.

To test your connection and query:

1. Open the Autonomous REST Connector Configuration Manager by selecting the ODBC Administrator from the Progress DataDirect group.

   For detailed information on launching the Configuration Manager, see "Configuring data sources with the Configuration Manager."

2. If you are not testing an existing data source, provide connection information using one of the following methods:

   - Enter a connection string you provide by clicking the Edit button ( ); then, pasting your string into the Connection String field. If you prefer, you can also type a string directly into this field.
   - Enter values of the connection options you want to configure into the corresponding fields. The Autonomous REST Connector Configuration Manager will generate a data source and connection string based on the values you specify.

3. Click **Test Connect** to attempt to connect to the service using the string specified in the Connection String field. The **Test Connection** window appears.

4. Provide connection option values for any fields required by your service.
5. Optionally, to execute a test query with the test connection, enter a SQL query into the Test Query field. For example:

```
SELECT * FROM INFORMATION_SCHEMA.SYSTEM_TABLES
```

6. Click **Execute**.

   If successful, the window displays a confirmation message and, if a query was specified, the results of the query.

See also

Configuring data sources with the Configuration Manager on page 80

Password Encryption Tool (UNIX/Linux only)

On UNIX and Linux, Progress DataDirect provides a Password Encryption Tool, called `ddencpwd`, that encrypts passwords for secure handling in connection strings and `odbc.ini` files. At connection, the driver decrypts these passwords and passes them to the data source as required. Passwords can be encrypted for any option, including:

- KeyPassword
- KeyStorePassword
- TrustStorePassword
- Password

To use the Password Encryption Tool:

1. From a command line, navigate to the directory containing the `ddencpwd` application. By default, this is *`install_directory`*`/tools`.
2. Enter the following command:

   ```
   ddencpwd password
   ```

   where:

   `password`

   is the password you want to encrypt.

3. The tool returns an encrypted password value. Specify the returned value for the corresponding attribute in the connection string or `odbc.ini` file. For example, if you encrypted the password for `KeyPassword`, specify the following in your connection string or datasource definition:

   ```
   KeyPassword=returned_value
   ```

4. Repeat Steps 2 and 3 to encrypt additional passwords.
5. If using an `odbc.ini` file, save your file.

This completes this tutorial. You are now ready to connect using encrypted passwords.

Using a logon dialog box

Some ODBC applications display a logon dialog box when you are connecting to a data source. The fields exposed in the logon dialog depend on the setting of the Authentication Method option. To connect, provide the values described in the following sections; then, click **OK** to complete the logon.

Basic authentication (AuthenticationMethod=0)

In the dialog box, provide the following information:

- In the User Name field, type your logon ID.
- In the Password, type your password.

Bearer Token authentication (AuthenticationMethod=34)

In the dialog box, provide the following information:

- In the Security Token field, enter your API token to be passed as a bearer token.

Digest authentication (AuthenticationMethod=30)

In the dialog box, provide the following information:

- In the User Name field, type your logon ID.
- In the Password, type your password.

HTTP header authentication (AuthenticationMethod=25)

In the dialog box, provide the following information:

- In the AuthHeader field, type the name of the HTTP header used for authentication. The default is `Authorization`.
- In the SecurityToken field, type to the security token required to make a connection to your endpoint. For example, `XaBARTsLZReM`.

URL parameter authentication (AuthenticationMethod=14)

In the dialog box, provide the following information:

- In the AuthParam field, type the name of the URL parameter used to pass the security token. For example, `apikey`.
- In the SecurityToken field, specify the security token required to make a connection to your endpoint. For example, `XaBARTsLZReM`.

Using IP addresses

The driver supports Internet Protocol (IP) addresses in the IPv4 and IPv6 formats.

If your network supports named servers, the server name specified in the data source can resolve to an IPv4 or IPv6 address. Addresses can be specified using the REST Sample Path option or the Model file specified by the REST Config File option.

In the following connection string example, the IP address for the REST service is specified in IPv4 format using the REST Sample Path option:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;RestSamplePath='123.456.78.90'
```

In the following connection string example, the IP address for the REST service is specified in IPv6 format using the REST Sample Path option:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;
       RestSamplePath='2001:DB8:0000:0000:8:800:200C:417A'
```

In addition to the normal IPv6 format, the drivers in the preceding tables support IPv6 alternative formats for compressed addresses. For example, the following connection string specifies the server using IPv6 format, but uses the compressed syntax for strings of zero bits:

```
DRIVER=DataDirect 8.0 Autnomous REST
Connector;RestSamplePath='2001:DB8:0:0:8:800:200C:417A'
```

For complete information about IPv6 formats, go to the following URL:

http://tools.ietf.org/html/rfc4291#section-2.2

Authentication

The driver supports the following authentication methods:

- *No Authentication* is used for REST services that do not require authentication.
- *Basic Authentication* uses user IDs, passwords, and HTTP headers to authenticate.
- *Bearer token Authentication* authenticates an API Token, configured as BearerToken.
- *Digest Authentication* negotiates user ID and password authentication using digest access authentication.
- *HTTP Header Authentication* passes security tokens via the HTTP headers to authenticate.
- *URL Parameter Authentication* authenticates by passing security tokens using URLs.
- *OAuth 2.0 Authentication* authentications using OAuth 2.0 authentication flows.
- *Custom Authentication* authenticates using a series of requests defined in the Model file.

By default, the driver is configured to use no authentication (`AuthenticationMethod=15`).

See also

Authentication Method on page 147

Basic authentication

To configure the driver to use basic authentication:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the REST Sample Path option, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `0` Basic.
- Set the AuthHeader (`AuthHeader`) option to specify the name of the HTTP header used for authentication. The default is `Authorization`.
- Set the User Name (`User`) option to specify your logon ID.
- Set the Password (`Password`) option to specify your password.
- Optionally, set the Health Check URI (`HealthURI`) option to specify the URI that the driver calls to confirm connectivity. Services using HTTP header authentication do not perform an explicit action upon connection. You can work around this limitation by specifying a value for this option. The driver performs a connectivity test at connection, when executing a test connect, or whenever the driver needs to confirm connectivity.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following examples demonstrate a session using a Model file with basic authentication enabled and AuthHeader set to the default:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=0;
       RestConfigFile=C:/path/to/myrest.rest;User=jsmith;Password=secret;
```

Using an `odbc.ini` file with the 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so
Description=My Autonomous REST Data Source
...
AuthenticationMethod=0
...
Password=secret
...
RestConfigFile=C:/path/to/myrest.rest
...
User=jsmith
...
```

See also

Connection option descriptions on page 133

AWS credentials authentication

To configure the driver to use AWS credentials authentication:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the REST Sample Path option, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `43` (AWS).
- Set the Access Key (`AccessKey`) option to specify your access key ID for your IAM user or AWS account root user.
- Set the Region (`Region`) option to specify name of the region that hosts your AWS server. For example, `us-east-1` or `us-east-2`. If no value is specified, the driver will use `us-east-1`.

  For a list of regions, refer to the AWS documentation.
- Set the Secret Key (`SecretKey`) option to specify your secret access key for an IAM user or AWS account root user.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following examples demonstrate a session using a Model file with AWS authentication enabled.

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AccessKey=ABCDEFGHIJKL1EXAMPLE;
  AuthenticationMethod=43;Region=us-east-2;RestConfigFile=C:/path/to/myrest.rest;
  SecretKey=aBcdeFGhiJKLM/N1OPQRS/tUvWxyzYEXAMPLEKEY
```

Using an `odbc.ini` file with the 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so
Description=My Autonomous REST Data Source
...
AccessKey=ABCDEFGHIJKL1EXAMPLE
...
AuthenticationMethod=43
...
Password=secret
...
Region=us-east-2
...
RestConfigFile=C:/path/to/myrest.rest
...
SecretKey=aBcdeFGhiJKLM/N1OPQRS/tUvWxyzYEXAMPLEKEY
...
User=jsmith
...
```

See also

Connection option descriptions on page 133

Bearer token authentication

To configure the driver to use bearer token authentication:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the REST Sample Path option, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `34` (Bearer token).
- Set the SecurityToken (`SecurityToken`) option to specify your the API Token, configured as BearerToken, used for authentication.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following examples demonstrate a session using a Model file with bearer token authentication and using a Model file:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=34;
       RestConfigFile=C:/path/to/myrest.rest;
       SecurityToken=C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx
```

Using an `odbc.ini` file with the 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so
Description=My Autonomous REST Data Source
...
AuthenticationMethod=34
...
RestConfigFile=C:/path/to/myrest.rest
...
```

See also

Connection option descriptions on page 133

Digest authentication

To configure the driver to use digest authentication:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the REST Sample Path option, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `30` (Digest).
- Set the User Name (`User`) option to specify your logon ID that is used to connect to your REST service. For example, `jsmith`.
- Set the Password (`Password`) option to specify your password.
- Optionally, set the Health Check URI (`HealthURI`) option to specify the URI that the driver calls to confirm connectivity. Services using HTTP header authentication do not perform an explicit action upon connection. You can work around this limitation by specifying a value for this option. The driver performs a connectivity test at connection, when executing a test connect, or whenever the driver needs to confirm connectivity.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following examples demonstrate a session using a Model file with digest authentication enabled and using a Model file:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=30;
       RestConfigFile=C:/path/to/myrest.rest;User=jsmith;Password=secret;
```

Using an `odbc.ini` file with the 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so
Description=My Autonomous REST Data Source
AuthenticationMethod=30
...
Password=secret
...
RestConfigFile=C:/path/to/myrest.rest
...
User=jsmith
...
```

See also

Connection option descriptions on page 133

HTTP header authentication

To configure the driver to use HTTP header authentication:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the REST Sample Path method, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `25` (HttpHeader).
- Set the AuthHeader (`AuthHeader`) option to specify the name of the HTTP header used for authentication. The default is `Authorization`.
- Set the Security Token (`SecurityToken`) option to specify the security token required to make a connection to your endpoint. For example, `XaBARTsLZReM`.
- Optionally, set the Health Check URI (`HealthURI`) option to specify the URI that the driver calls to confirm connectivity. Services using HTTP header authentication do not perform an explicit action upon connection. You can work around this limitation by specifying a value for this option. The driver performs a connectivity test at connection, when executing a test connect, or whenever the driver needs to confirm connectivity.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following examples demonstrates a session using a Model file with HTTP header authentication enabled and AuthHeader is set to the default.

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=25;
       RestConfigFile=C:/path/to/myrest.rest;SecurityToken=XaBARTsLZReM;
```

Using an `odbc.ini` file with the 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so
Description=My Autonomous REST Data Source
...
AuthenticationMethod=25
...
RestConfigFile=C:/path/to/myrest.rest
...
SecurityToken=XaBARTsLZReM
...
```

See also

Connection option descriptions on page 133

URL parameter authentication

To configure the driver to use URL parameter authentication:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the REST Sample Path method, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `14` (UrlParameter).
- Set the AuthParam (`AuthParam`) option to specify the name of the URL parameter used to pass the security token. For example, `apikey`.
- Set the Security Token (`SecurityToken`) option to specify the security token required to make a connection to your endpoint. For example, `XaBARTsLZReM`.
- Optionally, set the Health Check URI (`HealthURI`) option to specify the URI that the driver calls to confirm connectivity. Services using HTTP header authentication do not perform an explicit action upon connection. You can work around this limitation by specifying a value for this option. The driver performs a connectivity test at connection, when executing a test connect, or whenever the driver needs to confirm connectivity.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following examples demonstrates a session using a Model file with URL parameter authentication enabled.

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=14;
       AuthParam=apikey;RestConfigFile=C:/path/to/myrest.rest;
       SecurityToken=XaBARTsLZReM;
```

Using a `odbc.ini` file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
...
Description=My Autonomous REST Data Source
...
AuthenticationMethod=14
...
AuthParam=apikey
...
RestConfigFile=C:/path/to/myrest.rest
...
SecurityToken=XaBARTsLZReM
...
```

See also

Connection option descriptions on page 133

OAuth 2.0 authentication

OAuth 2.0 is an authentication protocol that is commonly used by REST services and websites to authorize access to their data. While OAuth 2.0 offers a number of benefits, including the ability to limit the scope of access privileges and support for multiple points of authentication, its primary advantage is that it allows for access delegation without the issuance of passwords. Instead, the protocol relies on the distribution of temporary access tokens to verify that an application is authorized to access data stored on the site.

Although access tokens ultimately grant access privileges to endpoints that use OAuth 2.0 authentication, there are multiple authentication flows that you can use to obtain them. These authentication flows, or grant types, differ based on environment and security needs of the site. Because of this, each grant type requires a different set of credentials and authentication endpoints to successfully authenticate. The following sections describe some common grant types and their required options. Note that your authentication flow may differ from the types listed here. If you are unsure of your requirements, contact your system administrator.

Access token flow

The access token authentication flow employs a simple authentication flow that passes the access token directly from the client to the REST service for authentication. The access token is obtained from external resource or from tools, such as the Configuration Manager or Postman, and specified using the AccessToken property.

Note: As opposed to using a third-party application such as Postman, you can use the Progress DataDirect Autonomous REST Connector Configuration Manager to obtain an access token to support the access token flow. See Obtaining access and refresh tokens using the Configuration Manager on page 117 for details.

To configure the driver to use an access token authentication flow:

- The application should be configured to set the Access Token (`AccessToken`) option to specify the access token required to authenticate to a REST service.

  Note: Access tokens typically expire ten minutes after generation. Once connected, the access token remains valid till the session is disconnected.

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/yelp.rest`.
  - If you are using the REST Sample Path method, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `50` (OAuth2-Access Token).

  Note: To support existing configurations, the Authentication Method option will continue to support the `24` (OAuth2) value for the access token flow.

- Optionally, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
  - Set the Authentication Header (`AuthHeader`) option to specify the name of the HTTP header used for authentication.
  - Set the Security Token (`SecurityToken`) option to specify the value of the HTTP header named by the Authentication Header option.

  For example, if you have a header value of `Authorization:1a2bc34def567`, you would specify a values of `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.

  Note: You can specify multiple custom HTTP headers using the `#headers` in the Model file. See "Requests with custom HTTP headers" for details.

The following example demonstrates a basic Yelp session using an access token flow:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;
       AccessToken=C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx;
       AuthenticationMethod=50;RestConfigFile=C:/path/to/yelp.rest;
```

Using an `odbc.ini` file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
Description=My Autonomous REST Data Source
...
AuthenticationMethod=50
...
RestConfigFile=C:/path/to/yelp.rest
...
```

Note: The AccessToken option is not usually defined in data source definitions due to the short lifespan of access tokens.

See also

Requests with custom HTTP headers on page 228
Connection option descriptions on page 133

Authorization code grant

The authorization code grant is a commonly used authorization flow for web and native applications. It provides secure connections by requiring multiple points of authentication before permitting access to data. When using the authorization code flow, the application first navigates to the location hosting the temporary authorization code and retrieves it. Next, the authorization code is exchanged for an access token from the location specified in the Token URI option. If authentication takes place with a third-party authentication service, the application is redirected to the endpoint provided in the Redirect URI option to begin the session.

To use an authorization code grant:

- The application should be configured to set the Auth Code (`OauthCode`) option to specify the authorization code that is exchanged for the access token.
- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/box.rest`.
  - If you are using the REST Sample Path method, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `51` (OAuth2-Authorization Code).

  Note: To support existing configurations, the Authentication Method option will continue to support the `24` (OAuth2) value for the authorization code grant.

- Set the Client ID (`ClientID`) option to specify the client ID key for your application.
- Set the Token URI (`TokenURI`) option to specify the endpoint used to exchange authentication credentials for access tokens.

  Note: By default, the connector prefixes the token URI endpoint with a GET request method. However, some OAuth implementations require that the token URI endpoint be passed with a POST request method. In this scenario, the token URI endpoint must be prefixed with POST when specifying the value of the TokenURI option. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- If required by your authentication flow, set the Redirect URI (`RedirectURI`) option to specify the endpoint that the client is returned to after authenticating with a third-party service.
- Optionally, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
  - Set the Authentication Header (`AuthHeader`) option to specify the name of the HTTP header used for authentication.
  - Set the Security Token (`SecurityToken`) option to specify the value of the HTTP header named by the Authentication Header option.

  For example, if you have a header value of `Authorization:1a2bc34def567`, you would specify a values of `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.

  Note: You can specify multiple custom HTTP headers using the `#headers` in the Model file. See "Requests with custom HTTP headers" for details.

- Optionally, set the Scope (`Scope`) option to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the OAuth Client Credentials Mode (`ClientCredentialsMode`) option to determine how client credentials are sent in a request to obtain an access token . Configure this option for flows that require client credentials to be specified as only a basic authentication header or only as a URL parameter.
  - If set to `0` (Default), the client credentials are sent as a basic authentication header. This is the default setting.

   - If set to 1 (Basic), the client credentials are sent as a basic authentication header.
   - If set to 2 (Url), the client credentials are sent as a URL parameter.
   - If set to 3 (Post), the client credentials are sent in the body of a POST request.

- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following example demonstrates a basic session for a Box account using an authorization code grant:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=51;
       OauthCode=xyz123abc;ClientID=abcdefghij1k2lmn3o4p5qr67s;
       RestConfigFile=C:/path/to/box.rest;TokenURI=https://api.box.com/oauth2/token;
```

Using an odbc.ini file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
Description=My Autonomous REST Data Source
...
AuthenticationMethod=51
...
ClientID=abcdefghij1k2lmn3o4p5qr67s
...
RestConfigFile=C:/path/to/box.rest
...
TokenURI=https://api.box.com/oauth2/token
...
```

Note: The OAuthCode option is not typically defined in a data source definitions due to the short lifespan of authorization codes.

See also

Requests with custom HTTP headers on page 228
Connection option descriptions on page 133

Client credentials grant

The authentication flow for the client credentials grant exchanges client credentials for the access token at the location specified by the Token URI option.

To configure the driver to use a client credentials grant:

- Configure the minimum options required for a connection:
   - If you are using a Model file, set the REST Config File (RestConfigFile) option to provide the name and location of the Model file. For example, C:/path/to/googleanalytics.rest.
   - If you are using the REST Sample Path method, set the REST Sample Path (RestSamplePath) option to specify the endpoint that you want to connect to and sample. For example, https://example.com/countries/.

- Set the Authentication Method (AuthenticationMethod) option to 53 (OAuth2-Client Credentials).

Note: To support existing configurations, the Authentication Method option will continue to support the `24` (OAuth2) value for the client credentials grant.

- Set the Client ID (`ClientID`) option to specify the client ID key for your application.
- Set the Client Secret (`ClientSecret`) option to specify client secret for your application.

Important: The client secret is a confidential value used to authenticate the application to the server. To prevent unauthorized access, this value must be securely maintained.

- Set the Token URI (`TokenURI`) option to specify the endpoint used to exchange authentication credentials for access tokens.

Note: By default, the connector prefixes the token URI endpoint with a GET request method. However, some OAuth implementations require that the token URI endpoint be passed with a POST request method. In this scenario, the token URI endpoint must be prefixed with POST when specifying the value of the TokenURI option. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- Optionally, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
  - Set the Authentication Header (`AuthHeader`) option to specify the name of the HTTP header used for authentication.
  - Set the Security Token (`SecurityToken`) option to specify the value of the HTTP header named by the Authentication Header option.

  For example, if you have a header value of `Authorization:1a2bc34def567`, you would specify a values of `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.

Note: You can specify multiple custom HTTP headers using the `#headers` in the Model file. See "Requests with custom HTTP headers" for details.

- Optionally, set the Scope (`Scope`) option to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the OAuth Client Credentials Mode (`ClientCredentialsMode`) option to determine how client credentials are sent in a request to obtain an access token . Configure this option for flows that require client credentials to be specified as only a basic authentication header or only as a URL parameter.
  - If set to `0` (Default), the client credentials are sent as a basic authentication header. This is the default setting.
  - If set to `1` (Basic), the client credentials are sent as a basic authentication header.
  - If set to `2` (Url), the client credentials are sent as a URL parameter.
  - If set to `3` (Post), the client credentials are sent in the body of a POST request.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following example demonstrates a basic Google Analytics session using a client credentials grant:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=53;
       ClientID=123456789876-a1bc2de3fgh4ij567klmn8opqr.apps.googleusercontent.com;
       ClientSecret=FaZBFRsGXTaR;RestConfigFile=C:/path/to/googleanalytics.rest;
       TokenURI=https://accounts.google.com/o/oauth2/token;
```

Using an `odbc.ini` file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
Description=My Autonomous REST Data Source
...
AuthenticationMethod=53
...
ClientID=123456789876-a1bc2de3fgh4ij567klmn8opqr9.apps.googleusercontent.com
...
ClientSecret=FaZBFRsGXTaR
...
RestConfigFile=C:/path/to/googleanalytics.rest
...
TokenURI=https://accounts.google.com/o/oauth2/token
...
```

See also

Requests with custom HTTP headers on page 228
Connection option descriptions on page 133

Dynamic authorization code grant

Dynamic authorization code grant allows you to initiate an authorization code grant flow by specifying login credentials using the login prompt for your REST service, thereby providing a method to authenticate without fetching access and refresh tokens via the Configuration Manager or third-party application. Similar to authorization code grant, dynamic authorization code grant is typically used for web and native applications. It also provides secure connections by requiring multiple points of authentication before permitting access to data.

When connecting with dynamic authorization code grant flow, the driver launches the login prompt for your service in a separate browser window. After you submit your user and password credentials via the prompt, the driver exchanges your login credentials and client credentials for the Authorization Code from the location specified by the Authorization URI option. The driver then navigates to the endpoint specified by the Token URI option to exchange the authorization code for the access and refresh tokens. Finally, the application is redirected to the location provided in the Redirect Option to begin the session.

After the grant flow is complete, the driver continues to use the access token and refresh tokens to access data resources for the lifetime of the ODBC connection or until both the access and refresh token expires, whichever occurs first. If both tokens expire while the connection is still active, the driver will launch the login prompt to reinitiate the flow.

Note: The dynamic authorization grant requires the manual submission of login credentials via the login prompt for your service; therefore, the driver does not support dynamic authorization grant in headless environments.

To use an dynamic authorization code grant:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/box.rest`.
  - If you are using the REST Sample Path method, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `51` (OAuth2-Authorization Code).

Note: To support existing configurations, the Authentication Method option will continue to support the `24` (OAuth2) value for the dynamic authorization code grant.

- Set the Enable Login Prompt (`EnableLoginPrompt`) option to `1` (enabled). When Enable Login Prompt is enabled, the driver launches the login prompt for your service in a separate browser window to initiate the OAuth grant flow.
- Set the SQL Engine Mode (`SQLEngineMode`) option to `2` (Direct). Note that this is the default setting for Windows.

Note: The dynamic authorization code grant is supported only in Direct mode.

- Set the Client ID (`ClientID`) option to specify the client ID key for your application.
- Set the Client Secret (Client Secret) option to specify the client secret for your application.

Important: The client secret is a confidential value used to authenticate the application to the server. To prevent unauthorized access, this value must be securely maintained.

- Set the Authorization URI (`AuthURI`) option to specify the endpoint for obtaining an authorization code.
- Set the Token URI (`TokenURI`) option to specify the endpoint used to exchange authentication credentials for access tokens.

Note: By default, the connector prefixes the token URI endpoint with a GET request method. However, some OAuth implementations require that the token URI endpoint be passed with a POST request method. In this scenario, the token URI endpoint must be prefixed with POST when specifying the value of the TokenURI option. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- If required by your authentication flow, set the Redirect URI (`RedirectURI`) option to specify the endpoint that the client is returned to after authenticating with a third-party service. Note that the value of the Redirect URI (RedirectURI) option must include the port number. For example, `RedirectURI=http://localhost:80` or `RedirectURI=http://localhost:8080`.
- Optionally, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
  - Set the Authentication Header (`AuthHeader`) option to specify the name of the HTTP header used for authentication.
  - Set the Security Token (`SecurityToken`) option to specify the value of the HTTP header named by the Authentication Header option.

For example, if you have a header value of `Authorization:1a2bc34def567`, you would specify a values of `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.

Note: You can specify multiple custom HTTP headers using the `#headers` in the Model file. See "Requests with custom HTTP headers" for details.

- Optionally, set the Scope (`Scope`) option to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the OAuth Client Credentials Mode (`ClientCredentialsMode`) option to determine how client credentials are sent in a request to obtain an access token . Configure this option for flows that require client credentials to be specified as only a basic authentication header or only as a URL parameter.
  - If set to `0` (Default), the client credentials are sent as a basic authentication header. This is the default setting.
  - If set to `1` (Basic), the client credentials are sent as a basic authentication header.
  - If set to `2` (Url), the client credentials are sent as a URL parameter.
  - If set to `3` (Post), the client credentials are sent in the body of a POST request.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following example demonstrates a basic session for a Box account using an dynamic authorization code grant:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=51;
   AuthURI=https://api.box.com/oauth2/authorize;ClientID=abcdefghij1k2lmn3o4p5qr67s;
   ClientSecret=FaZBFRsGXTaR;EnableLoginPrompt=1;SQLEngineMode=2
   RedirectURI=https://localhost:80;RestConfigFile=C:/path/to/box.rest;
   SQLEngineMode=2;TokenURI=https://api.box.com/oauth2/token;
```

Using an `odbc.ini` file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
Description=My Autonomous REST Data Source
...
AuthenticationMethod=51
...
AuthURI=https://api.box.com/oauth2/authorize
...
ClientID=abcdefghij1k2lmn3o4p5qr67s
...
ClientSecret=FaZBFRsGXTaR;
...
EnableLoginPrompt=1
...
RedirectURI=https://localhost:80
...
RestConfigFile=C:/path/to/box.rest
...
SQLEngineMode=2;
...
TokenURI=https://api.box.com/oauth2/token
...
```

See also

Requests with custom HTTP headers on page 228

Connection option descriptions on page 133

JWT bearer grant

Prerequisites:

- A client application registered with the authorization service.
- A JWT certificate containing the private key for the registered application.

The JWT(JSON Web Token) bearer grant flow is used to retrieve access tokens without having to pass confidential credentials to an authorization provider. This is accomplished by leveraging independent security domains that have a trust relationship: an identity provider and an authorization server. The identity provider, which can be the client or a third-party service, generates the JWT token from specified credential information. The client can then exchange the JWT token for the access tokens from the authorization server.

To configure the driver to use an authentication flow for a refresh token grant:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/docusign.rest`.
  - If you are using the REST Sample Path method, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `55` (OAuth2-JWT Bearer).

  Note: To support existing configurations, the Authentication Method option will continue to support the `24` (OAuth2) value for the JWT bearer grant flow.

- Set the Claims Issuer (`ClaimsIssuer`) option to specify the client ID or consumer key of the authorization server.
- Set the Claims Subject (`ClaimsSubject`) option to specify your username.
- Set the JWT Certification Store (`JWTCertStore`) option to specify the file path of the certificate store containing the private key used for JWT authentication.
- If required by your grant flow, set the JWT Certification Password (`JWTCertPassword`) option to specify the password for the JWT certificate.
- Optionally, set the JWT Certification Alias (`JWTCertAlias`) option to specify an alias for the JWT certificate.
- If required by your grant flow, set the Token URI (`TokenURI`) option to specify the endpoint used to exchange authentication credentials for access tokens.

  Note: By default, the connector prefixes the token URI endpoint with a GET request method. However, some OAuth implementations require that the token URI endpoint be passed with a POST request method. In this scenario, the token URI endpoint must be prefixed with POST when specifying the value of the TokenURI option. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- If required by your authentication flow, set the Redirect URI (`RedirectURI`) option to specify the endpoint that the client is returned to after authenticating with a third-party service. Note that the value of the Redirect URI (RedirectURI) option must include the port number. For example, `RedirectURI=http://localhost:80` or `RedirectURI=http://localhost:8080`.
- If required by your grant flow, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
  - Set the Authentication Header (`AuthHeader`) option to specify the name of the HTTP header used for authentication.
  - Set the Security Token (`SecurityToken`) option to specify the value of the HTTP header named by the Authentication Header option.

  For example, if you have a header value of `Authorization:1a2bc34def567`, you would specify a values of `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.

  Note: You can specify multiple custom HTTP headers using the `#headers` in the Model file. See "Requests with custom HTTP headers" for details.

- Optionally, set the Scope (`Scope`) option specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.

The following example demonstrates a simple configuration for DocuSign using a JWT bearer grant. Note that DocuSign requires you to request application consent before using JWT authentication. After providing the following values, you can use the **Fetch OAuth Token** button on the Configuration Manager to fetch the application consent:

- Client ID
- Client secret
- Auth URI

Refer to the DocuSign documentation for more information and the latest requirements.

Using a connection string:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;
       RestConfigFile=C:/path/to/docusign.rest;AuthenticationMethod=55;
       ClaimsIssuer='1a2b3c4d5e_6f7g8h9g';ClaimsSubject=jsmith@example.com;
       JWTCertStore=jwtcert.jks;JWTCertPassword=secret;
       TokenUri=https://account-d.docusign.com/oauth/token;
       RedirectUri=http://localhost:3000;AuthHeader=response_type;SecurityToken=code;
       Scope=signature impersonation;
```

Using an `odbc.ini` file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
Description=My Autonomous REST Data Source
...
AuthenticationMethod=55
...
AuthHeader=response_type
...
ClaimsIssuer='1a2b3c4d5e_6f7g8h9g'
...
ClaimsSubject=jsmith@example.com
...
JWTCertStore=jwtcert.jks
...
RedirectUri=http://localhost:3000
...
RestConfigFile=C:/path/to/docusign.rest
...
TokenUri=https://account-d.docusign.com/oauth/token
...
Scope=signature impersonation
...
SecurityToken=code
...
```

See also

Connection option descriptions on page 133

Password grant

The authentication flow for the password grant exchanges user credentials for the access token at the location specified by the Token URI option. For added security, client credentials, such as the client ID and client secret, might also be authenticated for some flows.

To configure the driver to use an authentication flow for a password grant:

- Configure the minimum required options required for a connection:

    - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/zendesk.rest`.
    - If you are using the REST Sample Path method, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.

- Set the Authentication Method (`AuthenticationMethod`) option to `52` (OAuth2-Password).

Note: To support existing configurations, the Authentication Method option will continue to support the `24` (OAuth2) value for the password grant.

- Set the User (`User`) option to specify the user name that is used to fetch the access token from the Token endpoint.
- Set the Password (`Password`) option to specify the password used to fetch the access token.
- Set the Token URI (`TokenURI`) option to specify the endpoint used to exchange authentication credentials for access tokens.

Note: By default, the connector prefixes the token URI endpoint with a GET request method. However, some OAuth implementations require that the token URI endpoint be passed with a POST request method. In this scenario, the token URI endpoint must be prefixed with POST when specifying the value of the TokenURI option. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- If required by your REST service, set the Client ID (`ClientID`) option to specify the client ID key for your application.
- If required by your REST service, set the Client Secret (`ClientSecret`) option to specify the client secret for your application.

Important: The client secret is a confidential value used to authenticate the application to the server. To prevent unauthorized access, this value must be securely maintained.

- Optionally, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
    - Set the Authentication Header (`AuthHeader`) option to specify the name of the HTTP header used for authentication.
    - Set the Security Token (`SecurityToken`) option to specify the value of the HTTP header named by the Authentication Header option.

  For example, if you have a header value of `Authorization:1a2bc34def567`, you would specify a values of `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.

Note: You can specify multiple custom HTTP headers using the `#headers` in the Model file. See "Requests with custom HTTP headers" for details.

- Optionally, set the Scope (`Scope`) option to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.

- Optionally, set the OAuth Client Credentials Mode (`ClientCredentialsMode`) option to determine how client credentials are sent in a request to obtain an access token . Configure this option for flows that require client credentials to be specified as only a basic authentication header or only as a URL parameter.
  - If set to `0` (Default), the client credentials are sent as a basic authentication header. This is the default setting.
  - If set to `1` (Basic), the client credentials are sent as a basic authentication header.
  - If set to `2` (Url), the client credentials are sent as a URL parameter.
  - If set to `3` (Post), the client credentials are sent in the body of a POST request.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following examples demonstrate a basic Zendesk session using a password grant:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=52;
       RestConfigFile=C:/path/to/zendesk.rest;
       TokenURI=https://accounts.google.com/o/oauth2/token;User=jjones@example.com;
       Password=secretstuff;
```

Using a `odbc.ini` file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
Description=My OAuth2 Autonomous REST Data Source
...
AuthenticationMethod=52
...
Password=secretstuff
...
RestConfigFile=C:/path/to/zendesk.rest
...
TokenURI=https://accounts.google.com/o/oauth2/token
...
User=jjones@example.com
...
```

See also

Requests with custom HTTP headers on page 228
Connection option descriptions on page 133

PKCE grant

PKCE (Proof Key for Code Exchange) authorization code grant is a more secure version of the standard authorization code grant type. To better protect against attacks, the PKCE flow also requires a client generated secret when exchanging the authorization code for the access token. This requirement prevents the access code from being acquired by malicious actors even if the authorization code is intercepted.

Note: The PKCE grant requires the manual submission of login credentials via the login prompt for your service; therefore, the driver does not support the PKCE grant type in headless environments.

To use PKCE authorization code grant:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/box.rest`.
  - If you are using the REST Sample Path method, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `56` (OAuth2-PKCE).
- Set the Client ID (`ClientID`) option to specify the client ID key for your application.
- Set the ClientSecret (`ClientSecret`) option to specify the client secret for your application.

  Note: The client secret is a confidential value used to authenticate the application to the server. To prevent unauthorized access, this value must be securely maintained.

- Set the Authorization URI (`AuthURI`) option to specify the endpoint for obtaining an authorization code.
- Set the Token URI (`TokenURI`) option to specify the endpoint used to exchange authentication credentials for access tokens.

  Note: By default, the connector prefixes the token URI endpoint with a GET request method. However, some OAuth implementations require that the token URI endpoint be passed with a POST request method. In this scenario, the token URI endpoint must be prefixed with POST when specifying the value of the TokenURI option. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- Set the Redirect URI (`RedirectURI`) option to specify the endpoint that the client is returned to after authenticating with a third-party service. Note that the value of the RedirectURI property must include the port number. For example, `RedirectURI=http://localhost:80` or `RedirectURI=http://localhost:8080`.
- Set the SQL Engine Mode (`SQLEngineMode`) option to `2` (Direct). Note that this is the default setting for Windows.

  Note: The dynamic authorization code grant is supported only in Direct mode.

- Optionally, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
  - Set the Authentication Header (`AuthHeader`) option to specify the name of the HTTP header used for authentication.
  - Set the Security Token (`SecurityToken`) option to specify the value of the HTTP header named by the Authentication Header option.

  For example, if you have a header value of `Authorization:1a2bc34def567`, you would specify a values of `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.

  Note: You can specify multiple custom HTTP headers using the `#headers` in the Model file. See "Requests with custom HTTP headers" for details.

- Optionally, set the Scope (`Scope`) option to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the OAuth Client Credentials Mode (`ClientCredentialsMode`) option to determine how client credentials are sent in a request to obtain an access token . Configure this option for flows that require client credentials to be specified as only a basic authentication header or only as a URL parameter.
  - If set to `0` (Default), the client credentials are sent as a basic authentication header. This is the default setting.
  - If set to `1` (Basic), the client credentials are sent as a basic authentication header.
  - If set to `2` (Url), the client credentials are sent as a URL parameter.
  - If set to `3` (Post), the client credentials are sent in the body of a POST request.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following example demonstrates a simple configuration for a Spotify™ service using PKCE grant:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;RestConfigFile=C:/path/to/box.rest;
       AuthenticationMethod=56;ClientID='abcdefghik2lmn3o5qr67s';
       ClientSecret=FaZBFRsGXTaR;AuthURI=https://accounts.spotify.com/authorize;
       RedirectURI=https://localhost:8080;
       TokenURI='https://accounts.spotify.com/api/token';SQLEngineMode=2
```

Using an `odbc.ini` file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
Description=My Autonomous REST Data Source
...
AuthenticationMethod=56
...
AuthURI=https://accounts.spotify.com/authorize
...
ClientID=abcdefghij1k2lmn3o4p5qr67s
...
ClientSecret=FaZBFRsGXTaR
...
RedirectURI=https://localhost:8080
...
RestConfigFile=C:/path/to/spotify.rest
...
SQLEngineMode=2
...
TokenURI=https://accounts.spotify.com/api/token
...
```

See also

Requests with custom HTTP headers on page 228
Connection option descriptions on page 133

Refresh token grant

The refresh token grant is used to replace expired access tokens with active ones by exchanging the refresh token at the endpoint specified by the Token URI option.

Note: As opposed to using a third-party application such as Postman, you can use the Progress DataDirect Autonomous REST Connector Configuration Manager to obtain an refresh token to support the refresh token grant. See Obtaining access and refresh tokens using the Configuration Manager on page 117 for details.

To configure the driver to use an authentication flow for a refresh token grant:

- Configure the minimum options required for a connection:
  - If you are using a Model file, set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/zendesk.rest`.
  - If you are using the REST Sample Path method, set the REST Sample Path (`RestSamplePath`) option to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the Authentication Method (`AuthenticationMethod`) option to `54` (OAuth2-Refresh Token).

  Note: To support existing configurations, the Authentication Method option will continue to support the `24` (OAuth2) value for the refresh token grant.

- Set the Client ID (`ClientID`) option to specify the client ID key for your application.
- Set the Client Secret (`ClientSecret`) option to specify the client secret for your application.

  Important: The client secret is a confidential value used to authenticate the application to the server. To prevent unauthorized access, this value must be securely maintained.

- Set the Refresh Token (`RefreshToken`) option to specify the refresh token used to request a new access token or renew an expired one.

  Important: The refresh token is a confidential value used to authenticate to the server. To prevent unauthorized access, this value must be securely maintained.

- Set the Token URI (`TokenURI`) option to specify the endpoint from which the driver fetches access tokens.

  Note: By default, the connector prefixes the token URI endpoint with a GET request method. However, some OAuth implementations require that the token URI endpoint be passed with a POST request method. In this scenario, the token URI endpoint must be prefixed with POST when specifying the value of the TokenURI option. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- Optionally, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
  - Set the Authentication Header (`AuthHeader`) option to specify the name of the HTTP header used for authentication.
  - Set the Security Token (`SecurityToken`) option to specify the value of the HTTP header named by the Authentication Header option.

  For example, if you have a header value of `Authorization:1a2bc34def567`, you would specify a values of `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.

  Note: You can specify multiple custom HTTP headers using the `#headers` in the Model file. See "Requests with custom HTTP headers" for details.

- Optionally, set the Scope (`Scope`) option specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the OAuth Client Credentials Mode (`ClientCredentialsMode`) option to determine how client credentials are sent in a request to obtain an access token . Configure this option for flows that require client credentials to be specified as only a basic authentication header or only as a URL parameter.
  - If set to `0` (Default), the client credentials are sent as a basic authentication header. This is the default setting.
  - If set to `1` (Basic), the client credentials are sent as a basic authentication header.
  - If set to `2` (Url), the client credentials are sent as a URL parameter.
  - If set to `3` (Post), the client credentials are sent in the body of a POST request.
- Optionally, specify values for any additional options you want to configure. See "Connection option descriptions" for a complete list of options.

The following examples demonstrate a Google Analytics session using a refresh token grant:

Using a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;AuthenticationMethod=54;
       ClientID=1234567898-a1bc2de3fgh4ij567klmn8opqr9stu.apps.googleusercontent.com
       Clientsecret=FaZBFRsGXTaR;RestConfigFile=C:/path/to/googleanalytics.rest;
       RefreshToken=1/abCD0F1GHijkLmNOPqrs_T2VWx3Y-Zabc45dE6FGh;
       TokenURI=https://accounts.google.com/o/oauth2/token;
```

Using an `odbc.ini` file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
Description=My Autonomous REST Data Source
...
AuthenticationMethod=54;
...
ClientID=1234567898-a1bc2de3fgh4ij567klmn8opqr9stu.apps.googleusercontent.com
...
Clientsecret=FaZBFRsGXTaR
...
RefreshToken=1/abCD0F1GHijkLmNOPqrs_T2VWx3Y-Zabc45dE6FGh
...
RestConfigFile=C:/path/to/googleanalytics.rest
...
TokenURI=https://accounts.google.com/o/oauth2/token
...
```

See also

Requests with custom HTTP headers on page 228
Connection option descriptions on page 133

Obtaining access and refresh tokens using the Configuration Manager

Note: The Configuration Manager is currently supported only on Windows platforms.

You need the following information before you begin.

- Sample endpoint: The endpoint of the service to which you are connecting
- Authorization URI: The endpoint for obtaining an authorization code from a third-party authorization service.

- Token URI: The endpoint used to exchange authentication credentials for access tokens.
- Client ID: The client ID for your application.
- Client Secret: The client secret for your application.
- Redirect URI: The endpoint to which the client is returned after third-party authorization for OAuth 2.0 implementations.
- Scope: An OAuth scope, or a space-separated list of OAuth scopes, which specifies the permissions that limit application access to the GitHub service.

Note: You can also use the Autonomous REST Composer (administrator view) to obtain refresh tokens as described in this section. For general information on launching the Autonomous REST Composer, see "Getting started with prebuilt Model files" or "Generating a Model file with the Autonomous REST Composer."

The following steps describe how you can use the Progress DataDirect Autonomous REST Connector Configuration Manager to obtain an access and refresh tokens for either the access token flow or the refresh token grant. In addition, the Configuration Manager produces a connection string that you can use in your application.

Note: You must allow popups in your browser to obtain access token with the Configuration Manager.

---

1. Open the Configuration Manager by selecting ODBC Administrator from the Progress DataDirect program group:

   For detailed information on launching the Configuration Manager, see "Configuring data sources with the Configuration Manager."

2. Set **Authentication Method** to `24` (OAuth2).
3. Provide the following information in the fields provided.
   - **REST Sample Path**
   - **Authorization URI**
   - **Token URI**
   - **Client ID**
   - **Client Secret**
   - **Scope** (if required by your REST service)

4. Retrieve the access token.
   a) Click **Fetch OAuth Token**.
   b) If the logon popup appears, enter your credentials. (This popup may not appear if you previously logged on.)
   c) If consent popup appears, provide consent, allowing the Configuration Manager to retrieve the token. (This popup may not appear if you previously provided consent to the Configuration Manager.)
   d) The **Access Token** and **Refresh Token** fields populate with the value retrieved from the OAuth authorization server.

5. Click **Test Connect** to verify connectivity and run SQL queries against the service.

Results:

The **Access Token** and **Refresh Token** fields include the access tokens refresh tokens that you can use to implement OAuth 2.0.

The connection string in the **Connection String** field may be copied and used in your application to connect with your REST service.

Note:

Not all the values in the resulting connection string are required. However, the connection string can be copied directly to a location that can be used by your application. The driver ignores any values that do not apply to your OAuth implementation.

For example, the refresh token grant connection string, derived from the Configuration Manager, might include the following options.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;RESTConfigFile=filepath;
      AuthenticationMethod=24;AuthURI=auth_uri;TokenURI=token_uri;ClientID=client_id;

       ClientSecret=client_secret;AccessToken=access_token;Scope=scope;
```

However, only the following properties are required for an refresh token grant connection string.

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;RESTConfigFile=filepath;
       AuthenticationMethod=24;TokenURI=token_uri;ClientID=client_id;
       ClientSecret=client_secret;AccessToken=access_token;
```

Custom authentication

If your service does not support one of the standard authentication methods provided by the driver, you can define a custom authentication requests using the Model file.

Before you start: Define your custom authentication requests in the Model file. For details, see "Custom authentication requests."

To configure the driver to use custom authentication requests:

- Set the REST Config File (`RestConfigFile`) option to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
- Set the Authentication Method (`AuthenticationMethod`) option to `29` (Custom). Note that `29` is the default when a custom authentication request is defined in the Model file.
- Optionally, set the Custom Auth Params (`CustomAuthParams`) option to specify the list of parameters to be used by the authentication requests defined in the Model file. Note that these values must be specified in the order that corresponds to the index location cited by the variable in the Model file. For example, the following `customAuthParams` variable points to the second (`2`) index:

  ```
  "company": "{customAuthParams[2]}"
  ```

  To successfully authenticate, the second value specified should be the value for the company field:

  ```
  CustomAuthParams=123XYZ456abc789;My Company Inc;www.example.com
  ```

- If applicable, set the Host Name (`HostName`) option to specify the host name portion of the HTTP endpoint to which you send requests. This value corresponds to the `[ServerName]` variable in the Model file.
- If required by your service, set the User (`User`) option to specify your logon ID. This value corresponds to the `[User]` variable in the Model file.
- If required by your service, set the Password (`Password`) option to specify your password. This value corresponds to the `[Password]` variable in the Model file.
- Optionally, specify values for any additional options you want to configure.

The following examples demonstrate sessions using custom authentication, including the optional CustomAuthParams, HostName, User, and Password options.

For a connection URL:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;CustomAuthParams=123XYZ456abc789;
       My Company Inc;path/to/endpoint;HostName=https://example.com;
       RestConfigFile=C:/path/to/myrest.rest;User=jsmith;Password=secret"
```

Using a `odbc.ini` file with a 32-bit driver:

```
Driver=ODBCHOME/lib/ivautorestxx.so;
Description=My Autonomous REST Data Source
...
CustomAuthParams=123XYZ456abc789;My Company Inc;path/to/endpoint;
...
HostName=https://example.com
...
Password=secret
...
RestConfigFile=C:/path/to/myrest.rest
...
User=jsmith
...
```

See also

Custom authentication requests on page 206

TLS/SSL encryption

The driver supports TLS/SSL data encryption. TLS/SSL works by allowing the client and server to send each other encrypted data that only they can decrypt. SSL negotiates the terms of the encryption in a sequence of events known as the *handshake*. During the handshake, the driver negotiates the highest TLS/SSL protocol available. The result of this negotiation determines the encryption cipher suite to be used for the TLS/SSL session.

The encryption cipher suite defines the type of encryption that is used for any data exchanged through a TLS/SSL connection. Some cipher suites are very secure and, therefore, require more time and resources to encrypt and decrypt data, while others provide less security, but are also less resource intensive.

The handshake involves the following types of authentication:

- *TLS/SSL server authentication* requires the server to authenticate itself to the client.
- *TLS/SSL client authentication* is optional and requires the client to authenticate itself to the server after the server has authenticated itself to the client.

Note: The version of TLS/SSL that is used and which cryptographic algorithm is used depends on which JVM you are using. Refer to your JVM documentation for more information about its TLS/SSL support.

Certificates

TLS/SSL encryption requires the use of a digitally-signed document, an x.509 standard certificate, for authentication and the secure exchange of data. The purpose of this certificate is to tie the public key contained in the certificate securely to the person/company that holds the corresponding private key. Your Progress DataDirect *for* ODBC drivers supports many popular formats. Supported formats include:

- DER Encoded Binary X.509
- Base64 Encoded X.509
- PKCS #12 / Personal Information Exchange

TLS/SSL server authentication

When the client makes a connection request, the server presents its public certificate for the client to accept or deny. The client checks the issuer of the certificate against a list of trusted Certificate Authorities (CAs) that resides in an encrypted file on the client known as a *truststore*. If the certificate matches a trusted CA in the truststore, an encrypted connection is established between the client and server. If the certificate does not match, the connection fails and the driver generates an error.

Most truststores are password-protected. The driver must be able to locate the truststore and unlock the truststore with the appropriate password. Two connection options are available to the driver to provide this information: Trust Store (Truststore) and Trust Store Password (TruststorePassword). The value of Trust Store is a pathname that specifies the location of the truststore file. The value of Trust Store Password is the password required to access the contents of the truststore.

Alternatively, you can configure the driver to trust any certificate sent by the server, even if the issuer is not a trusted CA. Allowing a driver to trust any certificate sent from the server is useful in test environments because it eliminates the need to specify truststore information on each client in the test environment. Setting the Validate Server Certificate (ValidateServerCertificate) connection option to false allows the driver to accept any certificate returned from the server regardless of whether the issuer of the certificate is a trusted CA.

Finally, the connection option, Host Name In Certificate (HostNameInCertificate), allows an additional method of server verification. When a value is specified for Host Name In Certificate, it must match the common name of the host in the Subject of the certificate. This prevents malicious intervention between the client and the server and ensures that the driver is connecting to the server that was requested.

The following examples show how to configure the driver to use data encryption via the SSL server authentication. In this configuration, since `ValidateServerCertificate=1`, the driver validates the certificate sent by the server and the host name specified by the HostNameInCertificate option.

Using a connection string:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;
       RESTConfigFile=/users/jsmith/path/to/myrest.rest;EncryptionMethod=1;
       HostNameInCertificate=MySubjectAltName;Keystore=KeyStoreName;
       KeystorePassword=YourKSPassword;ValidateServerCertificate=1
```

Using an `odbc.ini` file:

```
Driver=ODBCHOME/lib/ivautorestxx.so
Description=DataDirect Autonomous REST Connector
...
EncryptionMethod=1
...
HostName=YourServer
HostNameInCertificate=MySubjectAltName
...
RESTConfigFile=/users/jsmith/path/to/myrest.rest
...
Truststore=TrustStoreName
TruststorePassword=TSXYZZY
...
ValidateServerCertificate=1
...
```

See also

Connection option descriptions on page 133

TLS/SSL client authentication

If the server is configured for TLS/SSL client authentication, the server asks the client to verify its identity after the server identity has been proven. Similar to server authentication, the client sends a public certificate to the server to accept or deny. The client stores its public certificate in an encrypted file known as a *keystore*. Public certificates are paired with a private key in the keystore. To send the public certificate, the driver must access the private key.

Like the truststore, most keystores are password-protected. The driver must be able to locate the keystore and unlock the keystore with the appropriate password. Two connection options are available to the driver to provide this information: Keystore (KeyStore) and Keystore Password (KeyStorePassword). The value of KeyStore is a pathname that specifies the location of the keystore file. The value of Keystore Password is the password required to access the keystore.

The private keys stored in a keystore can be individually password-protected. In many cases, the same password is used for access to both the keystore and to the individual keys in the keystore. It is possible, however, that the individual keys are protected by passwords different from the keystore password. The driver needs to know the password for an individual key to be able to retrieve it from the keystore. An additional connection option, Key Password (KeyPassword), allows you to specify a password for an individual key.

The following examples show how to configure the driver to use data encryption via the SSL client authentication. In this configuration, since `ValidateServerCertificate=1`, the driver validates the certificate sent by the server and the host name specified by HostNameInCertificate.

Using a connection string:

```
DRIVER=DataDirect 8.0 Autonomous REST Connector;EncryptionMethod=1;
       RESTConfigFile=/users/jsmith/path/to/myrest.rest
       HostNameInCertificate=MySubjectAltName;Keystore=KeyStoreName;
       KeystorePassword=YourKSPassword;ValidateServerCertificate=1
```

Using the `odbc.ini` file:

```
Driver=ODBCHOME/lib/ivautorestxx.so
Description=DataDirect Autonomous REST Connector
...
EncryptionMethod=1
...
HostName=YourServer
HostNameInCertificate=MySubjectAltName
...
Truststore=TrustStoreName
TruststorePassword=TSXYZZY
...
RESTConfigFile=/users/jsmith/path/to/myrest.rest
...
ValidateServerCertificate=1
...
```

See also

Connection option descriptions on page 133

Connecting through a proxy server

In some environments, your application may need to connect through a proxy server, for example, if your application accesses an external resource such as a Web service. At a minimum, your application needs to provide the following connection information when you invoke the JVM if the application connects through a proxy server:

- Server name or IP address of the proxy server
- Port number on which the proxy server is listening for HTTP/HTTPS requests

In addition, if authentication is required, your application may need to provide a valid user ID and password for the proxy server. Consult with your system administrator for the required information.

For example, the following command invokes the JVM while specifying a proxy server named `pserver`, a port of `808`, and provides a user ID and password for authentication:

```
java -Dhttp.proxyHost=pserver -Dhttp.proxyPort=808 -Dhttp.proxyUser=smith
-Dhttp.proxyPassword=secret -cp autorest.jar com.acme.myapp.Main
```

Alternatively, you can use the Proxy Host (`ProxyHost`), Proxy Port (`ProxyPort`), Proxy User (`ProxyUser`), and Proxy Password (`ProxyPassword`) connection options. See "Connection Option Descriptions" for details about these attributes.

See also

Connection option descriptions on page 133

Performance considerations

Fetch Size/Web Service Fetch Size (FetchSize/WSFetchSize): The connection options Fetch Size and Web Service Fetch Size can be used to adjust the trade-off between throughput and response time. In general, setting larger values for Web Service Fetch Size and Fetch Size will improve throughput, but can reduce response time.

For example, if an application attempts to fetch 100,000 rows from the remote data source and Web Service Fetch Size is set to 500, the driver must make 200 Web service calls to get the 100,000 rows. If, however, Web Service Fetch Size is set to 4000, the driver only needs to make 25 Web service calls to retrieve 100,000 rows. Web service calls are expensive, so generally, minimizing Web service calls increases throughput. In addition, many Cloud data sources impose limits on the number of Web service calls that can be made in a given period of time. Minimizing the number of Web service calls used to fetch data also can help prevent exceeding the data source call limits.

For many applications, throughput is the primary performance measure, but for interactive applications, such as Web applications, response time (how fast the first set of data is returned) is more important than throughput. For example, suppose that you have a Web application that displays data 50 rows to a page and that, on average, you view three or four pages. Response time can be improved by setting Fetch Size to 50 (the number of rows displayed on a page) and WSFetch Size to 200. With these settings, the driver fetches all of the rows from the remote data source that you would typically view in a single Web service call and only processes the rows needed to display the first page.

Read Ahead (ReadAhead): The ReadAhead option allows you to issue multiple fetch requests in parallel. By increasing this number, you can improve throughput and performance, but it does so with the following restrictions:

- Larger values can increase the load on the server, which may adversely affect performance of other users. If you encounter issues, decrease the value specified for this option.
- Larger values may result in unnecessary requests if your application only requires the first few rows of results. This may be an issue if your service places limits on the number of web requests.

Caution: Due to potential impacts to other network and service users, we strongly recommend specifying only smaller values for the ReadAhead option. For example, in fully optimized environments, which include exceptionally fast connections and low latency, we recommend a setting of no higher than `5`. For typical environments, this value should be lower.

See also

Connection option descriptions on page 133

4

Using the SQL engine server

Some applications may experience problems loading the JVM required for the SQL engine because the process exceeds the available heap space. If your application experiences problems loading the JVM, you can configure the driver to operate in server mode.

The driver supports the following SQL engine modes:

- **Server mode**: The driver's SQL engine runs in a separate process with its own JVM, instead of trying to load the SQL engine and JVM in the same process used by the driver.
- **Direct mode**: The driver operates with the SQL engine and JVM running in a single process.
- **Auto mode**: The driver attempts to run in server mode. However, if server mode is unavailable, the SQL engine will failover to run in direct mode.

By default:

- **Windows**: The driver operates in server mode by default.
- **Linux**: The driver operates in direct mode by default.

Note: You must be an administrator to start or stop the service, or to configure any settings for the service.

See the following sections for details on configuring the SQL engine server on your platform.

For details, see the following topics:

- Configuring server mode using the Configuration Manager
- Configuring the SQL engine server using Java options
- Configuring Java logging for the SQL engine server

Configuring server mode using the Configuration Manager

In server mode, you must start the SQL engine server before connecting. Before starting the SQL engine server, choose a directory to store the local database files using the Schema Map (SchemaMap) option. Make sure that you have the correct permissions to write to this directory.

Note: The Configuration Manager is currently supported only on Windows platforms. To configure the SQL engine on Linux platforms, see "Configuring the SQL engine server using Java options."

The following sections describe how to configure, start, and stop the SQL engine server using the Configuration Manager.

1. Open your data source using the Configuration Manager; then, select the **SQL Engine** tab.
2. Set the SQL Engine Mode (SQLEngineMode) connection option to one of the following values:
   - **0 - Auto**: The SQL engine attempts to run in server mode first, but will failover to direct mode if server mode is unavailable.
   - **1 - Server**: The SQL engine runs exclusively in server mode.

   Note: By default, SQL Engine Mode is set to **1 - Server** for Windows and **2 - Direct** for Linux.

   The fields associated with server mode will become editable, and the **Start** button appears.
3. Provide values for the following options:
   - **Server Port Number**: Specifies a valid port on which the SQL engine listens for requests from the driver. The default value depends on your platform:

     32-bit: `19942`

     64-bit: `19941`
   - **JVM Classpath**: Specifies the CLASSPATH for the JVM used by the driver. See "JVM Classpath" for details. The default depends on your platform:

     Windows: `{.;c:\install_dir\java\lib\autorest.jar}`

     Linux: `{.:/home/user1/install_dir/java/lib/autorest.jar}`
   - **JVM Arguments**: A string that contains the arguments that are passed to the JVM that the driver is starting. The location of the JVM must be specified on your PATH. See "JVM Arguments" for details. The default value is:

     `-Xmx1024m`
   - **JVM Path**: Specifies fully qualified path to the JVM executable that you want to use to run the SQL engine server. The path must not contain double quotation marks.
4. Optionally, if connecting through a proxy server, provide values for the following options:
   - **Server Proxy Host**: Specifies the host name of the proxy server used by the SQL engine. The value specified can be a host name, a fully qualified domain name, or an IPv4 or IPv6 address.

- **Server Proxy Port**: Specifies the port needed to connect to the proxy server used by the SQL engine.
- **Server Proxy User**: Specifies the user name needed to connect to the proxy server used by the SQL engine.
- **Server Proxy Password**: Specifies the password needed to connect to the proxy server used by the SQL engine.

Note: After the initial configuration, in order for changes to the required and optional connection option values to take effect, you must restart the SQL engine server.

5. Click **Save** to save your changes
6. Click **Start** to run the SQL engine service. A message is displayed that indicates whether the SQL engine was able to successfully run.

See also

Configuring the SQL engine server using Java options on page 127
JVM Classpath on page 164
JVM Arguments on page 163

Stopping the SQL engine server using the Configuration Manager

To stop the SQL engine server:

1. Open the Configuration Manager and select the SQL Engine tab.
2. From the SQL Engine Mode drop-down list, select **0 - Auto** or **1 - Server**.
3. Click **Stop** to stop the service. A message is displayed to confirm that the service stopped.
4. Click **Exit** to close the Configuration Manager.

Configuring the SQL engine server using Java options

Before you start: In server mode, you must start the SQL engine server before using the driver. Before starting the SQL engine server, verify that you have the correct permissions to write to the directory specified by the Schema Map (SchemaMap) option.

The following sections describe how to configure, start, and stop the SQL engine server on Linux platform.

Note: By default, SQL Engine Mode is set to `1` (Server) for Windows and `2` (Direct) for Linux.

To configure the SQL engine server, specify values for the Java options in the following JVM argument to suit your environment. See the "SQL engine server Java options" table for a description of these options.

```
java -Xmx<heap_size>m -cp "<jvm_classpath>" com.ddtek.jdbc.<driver>.phoenix.sql.Server
      -port <port_number> -Dhttp.proxyHost=<proxy_host> -Dhttp.proxPort=<proxy_port>
       -Dhttp.proxyUser=<proxy_user> -Dhttp.proxyPassword=<proxy_password>
```

For example:

```
java -Xmx1024m -cp "/opt/Progress/DataDirect/ODBC_80_64bit/java/lib/autorest.jar"
        com.ddtek.phoenix.sql.Server -port 19941 -Dhttp.proxyHost=myhost@mydomain.com
        -Dhttp.proxPort=12345 -Dhttp.proxyUser=JohnQPublic -Dhttp.proxyPassword=secret
```

To start the SQL engine service, execute the JVM Argument after configuring the Java options. A confirmation message is returned once the server is online.

Note: After the initial configuration, in order for changes to these connection option values to take effect, you must restart the SQL engine server.

Table 11: SQL engine server Java options

Java Option	Description
Required Java Options	
`-cp`	Specifies the CLASSPATH for the Java Virtual Machine (JVM) used by the driver. The CLASSPATH is the search string the JVM uses to locate the Java jar files the driver needs. The driver's JVM is located on the following path: *install_dir*/java/lib/autorest.jar
`-port`	Specifies a valid port on which the SQL engine listens for requests from the driver. We recommend specifying one of the following values: • `19942` (32-bit drivers) • `19941` (64-bit drivers)
Optional Java Options	
`-Xmx`	Specifies the maximum memory heap size, in megabytes, for the JVM. The default size is determined by your JVM. We recommend specifying a size no smaller than `1024`. **Note:** Although this option is not required to start the SQL engine server, we highly recommend specifying a value.
`-Dhttp.proxyHost`	Specifies the host name of the proxy server. The value specified can be a host name, a fully qualified domain name, or an IPv4 or IPv6 address.
`-Dhttp.proxyPort`	Specifies the port number where the proxy server is listening for HTTP and/or HTTPS requests.
`-Dhttp.proxyUser`	Specifies the user name needed to connect to the proxy server.
`-Dhttp.proxyPassword`	Specifies the password needed to connect to the proxy server.

Stopping the SQL engine server

To stop the SQL engine server, choose one of the following:

- Using an application, execute `SHUTDOWN` SQL.
- From a command line, press `Ctrl` + `C`.

A message is returned to confirm that the service is stopped.

Configuring Java logging for the SQL engine server

Java logging for the SQL engine server can be configured using either the JVM or the driver.

For details, refer to "Configuring logging" in the *Progress DataDirect for ODBC Drivers Reference*.

5

Additional features and functionality

The following section describes additionally supported features and functionality that are specific to the driver.

For details, see the following topics:

- Identifiers
- Parameter metadata support

Identifiers

Identifiers are used to refer to objects exposed by the driver, such as tables and columns. The driver supports both unquoted and quoted identifiers for naming objects. An unquoted identifier must start with an ASCII alpha character and can be followed by zero or more ASCII alpha or numeric characters. Unquoted identifiers are converted to uppercase before being used.

Quoted identifiers must be enclosed in double quotation marks (""). A quoted identifier can contain any Unicode character, including the space character, and is case-sensitive. The driver recognizes the Unicode escape sequence \uxxxx as a Unicode character. You can specify a double quotation mark in a quoted identifier by escaping it with a double quotation mark.

The maximum length of both quoted and unquoted identifiers is 128 characters.

Note: When object names are passed as arguments to catalog functions, the case of the value must match the case of the name in the database. If an unquoted identifier name was used when the object was created, the value passed to the catalog function must be uppercase because unquoted identifiers are converted to uppercase before being used. If a quoted identifier name was used when the object was created, the value passed to the catalog function must match the case of the name as it was defined. Object names in results returned from catalog functions are returned in the case that they are stored in the database.

Parameter metadata support

The driver supports returning parameter metadata for Select statements that contain parameters in ANSI SQL 92 entry-level predicates, for example, such as COMPARISON, BETWEEN, IN, LIKE, and EXISTS predicate constructs. Refer to the ANSI SQL reference for detailed syntax.

Parameter metadata can be returned for a Select statement if one of the following conditions is true:

- The statement contains a predicate value expression that can be targeted against the source tables in the associated FROM clause. For example:

  ```
  SELECT * FROM foo WHERE bar > ?
  ```

  In this case, the value expression "bar" can be targeted against the table "foo" to determine the appropriate metadata for the parameter.

- The statement contains a predicate value expression part that is a nested query. The nested query's metadata must describe a single column. For example:

  ```
  SELECT * FROM foo WHERE (SELECT x FROM y    WHERE z = 1) < ?
  ```

The following Select statements show further examples for which parameter metadata can be returned:

```
SELECT col1, col2 FROM foo WHERE col1 = ? AND col2 > ?
        SELECT ... WHERE colname = (SELECT col2 FROM t2 WHERE col3 = ?)
        SELECT ... WHERE colname LIKE ?
        SELECT ... WHERE colname BETWEEN ? AND ?
        SELECT ... WHERE colname IN (?, ?, ?)
        SELECT ... WHERE EXISTS(SELECT ... FROM T2 WHERE col1 < ?)
```

ANSI SQL 92 entry-level predicates in a WHERE clause containing GROUP BY, HAVING, or ORDER BY statements are supported. For example:

```
SELECT * FROM t1 WHERE col = ? ORDER BY 1
```

Joins are supported. For example:

```
SELECT * FROM t1,t2 WHERE t1.col1 = ?
```

Fully qualified names and aliases are supported. For example:

```
SELECT a, b, c, d FROM T1 AS A, T2 AS B WHERE A.a = ? AND B.b = ?
```

6

Connection option descriptions

The connection option descriptions in this section are listed alphabetically by the GUI name that appears on the driver Setup dialog box. The connection string attribute name, along with its short name, is listed immediately underneath the GUI name in the option topics.

In most cases, the GUI name and the attribute name are the same; however, some exceptions exist. If you need to look up an option by its connection string attribute name, please refer to the following tables of connection string attribute names.

Also, a few connection string attributes, for example, Password, do not have equivalent options that appear on the GUI. They are in the list of descriptions by their attribute names.

Note: The driver does not support specifying values for the same connection option multiple times in a connection string or DSN. If a value is specified using the same attribute multiple times or using both long and short attributes, the connection may fail or the driver may not behave as intended.

The following tables provide a summary of supported connection options by functionality, including their attribute names, short names, and default values.

- General options
- Basic authentication options
- AWS credentials authentication options
- Bearer authentication options
- Custom authentication options
- Digest authentication options
- HTTP header authentication options
- OAuth 2.0 options
- URL parameter authentication options
- TSL/SSL data encryption options
- Mapping options
- SQL Engine options
- Proxy server options
- Web service options
- Additional options

General options

The following table summarizes general connection options that can apply to all connections that use data sources.

Table 12: General options

Attribute (Short Name)	Default
DataSourceName (dsn)	No default value
Description (desc)	No default value
HostName (host)	No default value
PortNumber (port)	SSL disabled: `80` SSL enabled: `443`
RESTConfigFile (rcf)	No default value
RESTSamplePath (rsp)	No default value

Basic (user ID and password) authentication options

The following table summarizes options used for basic authentication method.

Table 13: Basic authentication options

Attribute (Short Name)	Default
AuthenticationMethod (am)	`15` (None)

Attribute (Short Name)	Default
AuthHeader (ah)	`Authorization`
AuthParam (ap)	No default value
Password (pwd)	No default value
User	No default value

AWS credentials authentication options

The following table summarizes options used for AWS credentials authentication method.

Table 14: Basic authentication options

Attribute (Short Name)	Default
AccessKey (ak)	No default value
AuthenticationMethod (am)	`15` (None)
Region	No default value If no value is specified, the driver uses `us-east-1`.
SecretKey (sk)	No default value

Bearer token authentication options

The following table summarizes options used for bearer token authentication method.

Table 15: Bearer token authentication options

Attribute (Short Name)	Default
AuthenticationMethod (am)	`15` (None)
HealthURI (huri)	No default value
SecurityToken (st)	No default value

Custom authentication options

The following table summarizes connection options used for Custom authentication.

Table 16: Custom authentication optionss

Attribute (Short Name)	Default
AuthenticationMethod (am)	`15` (None)
CustomAuthParams (cap)	No default value

Attribute (Short Name)	Default
HealthURI (huri)	No default value
Password (pwd)	No default value
User	No default value

Digest authentication options

The following table summarizes options used for digest authentication.

Table 17: Digest authentication options

Attribute (Short Name)	Default
AuthenticationMethod (am)	`15` (None)
HealthURI (huri)	No default value
Password (pwd)	No default value
User	No default value

HTTP header authentication options

The following table summarizes options used for HTTP header authentication method.

Table 18: HTTP header authentication options

Attribute (Short Name)	Default
AuthenticationMethod (am)	`15` (None)
AuthHeader (ah)	`Authorization`
HealthURI (huri)	No default value
SecurityToken (st)	No default value

OAuth 2.0 options

The following table summarizes options used for OAuth 2.0 authentication. The OAuth 2.0 options you must specify depend on the grant type used in your environment. If you are unsure of the grant type or its requirements, contact your system administrator. For details on supported grant types, see OAuth 2.0 authentication.

Table 19: OAuth 2.0 options

Attribute (Short Name)	Default
AccessToken (atok)	No default value
AuthenticationMethod (am)	`15` (None)

Attribute (Short Name)	Default
AuthURI (o2au)	No default value
ClaimsIssuer (cliss)	No default value
ClaimsSubject (clsub)	No default value
ClientCredentialsMode (clcm)	0 (Default)
ClientID (cid)	No default value
ClientSecret (clse)	No default value
EnableLoginPrompt (elp)	0 (false)
HealthURI (huri)	No default value
JWTCertAlias (jwtca)	No default value
JWTCertPassword (jwtcp)	No default value
JWTCertStore (jwtcs)	No default value
LogoffURI (o2lu)	No default value
OAuthCode (oac)	No default value
RedirectURI (o2ru)	No default value
RefreshToken (rtok)	No default value
Scope (oas)	No default value
TokenURI (o2tu)	No default value

URL parameter authentication options

The following table summarizes connection options used for URL parameter authentication.

Table 20: URL parameter authentication options

Attribute (Short Name)	Default
AuthenticationMethod (am)	15 (None)
AuthParam (ap)	No default value
HealthURI (huri)	No default value

Attribute (Short Name)	Default
SecurityToken (st)	No default value
User	No default value

TSL/SSL data encryption options

The following table summarizes options used for TSL/SSL data encryption.

Table 21: TSL/SSL data encryption options

Attribute (Short Name)	Default
CryptoProtocolVersion (cpv)	No default value
EncryptionMethod (em)	`1` (SSL)
HostNameInCertificate (hnic)	No default value
KeyPassword (kp)	No default value
Keystore (ks)	No default value
KeystorePassword (ksp)	No default value
Truststore (ts)	No default value
TruststorePassword (tsp)	No default value
ValidateServerCertificate (vsc)	`1` (true)

Mapping options

The following table summarizes connection options involved in mapping the REST data model to a SQL model.

Table 22: Mapping options

Attribute (Short Name)	Default
ArrayNormalizationThreshold (art)	`12`
CreateMap (cm)	`3` (Session)
JSONRoot (jr)	No default value
QualifyNormalizedNames (qnn)	`0` (No)
RefreshSchema (rs)	`0` (false)
SchemaMap (smp)	Default value depends on the environment

SQL engine options

The following table lists the options used to configure the SQL engine.

Table 23: SQL Engine options

Attribute (Short Name)	Default
JVMArgs (jvma)	For the 32-bit driver when the SQL Engine Mode is set to 2 (Direct): -Xmx256m For all other configurations: -Xmx1024m
JVMClassPath (jvmcp)	No default value
JVMPath (jvmp)	*install_dir*\jre\bin\java.exe
ServerPortNumber (spn)	For 32-bit: 19952 For 64-bit: 19951
ServerProxyHost (sph)	No default value
ServerProxyPassword (spw)	No default value
ServerProxyPort (spp)	No default value
ServerProxyUser (spu)	No default value
SQLEngineMode (sem)	For Windows: 1 (Server) For Linux: 2 (Direct)
SQLService (ss)	No default value

Proxy server options

The following table summarizes proxy server connection options.

Table 24: Proxy server options

Attribute (Short Name)	Default
ProxyHost (pxhn)	No default value
ProxyPassword (pxpw)	No default value

Attribute (Short Name)	Default
ProxyPort (pxpt)	0 which means that the default value is determined by whether the value specified for the Proxy Host (ProxyHost) option is an HTTP or HTTPS URL. For HTTP: 80 For HTTPS: 443
ProxyUser (pxun)	No default value

Web service options

The following table summarizes Web service connection options, including those related to timeouts.

Table 25: Web service options

Attribute (Short Name)	Default
StmtCallLimit (scl)	0 (no limit)
StmtCallLimitBehavior (sclb)	1 (ErrorAlways)
WSFetchSize (wsfs)	2000 (rows)
WSPoolSize (wsps)	1
WSRetryCount (wsrc)	5
WSTimeout (wst)	120 (seconds)

Additional options

The following table summarizes additional connection options.

Table 26: Additional Options

Attribute (Short Name)	Default
DebugRecord (dbgrecord)	No default value
DefaultQueryOptions (dqo)	No default value
ExtendedOptions (xo)	No default value
FetchSize (fs)	100 (rows)
LogConfigFile (lgcf)	ddlogging.properties
ReadAhead (ra)	0

Attribute (Short Name)	Default
RefreshDirtyCache (rdc)	1 (true)
SamplingFailureTolerance (sft)	-1
Table (tbl)	No default value
TransactionMode (tm)	0 (NoTransactions)
UserAgent (ua)	No default value

For details, see the following topics:

- Access Key
- Access Token
- Array Normalization Threshold
- Authorization Code
- Authentication Header
- Authentication Method
- Authentication URL Parameter
- Authorization URI
- Claims Issuer
- Claims Subject
- Client ID
- Client Secret
- Create Map
- Crypto Protocol Version
- Custom Authentication Parameters
- Data Source Name
- Debug Folder
- Default Query Options
- Description
- Enable Login Prompt
- Encryption Method
- Extended Options
- Fetch Size

- Health Check URI
- Host Name
- Host Name In Certificate
- JSON Root
- JVM Arguments
- JVM Classpath
- JVM Path
- JWT Certificate Alias
- JWT Certificate Password
- JWT Certificate Store
- Key Password
- Keystore
- Keystore Password
- Log Config File
- Logoff URI
- OAuth Client Credentials Mode
- Password
- Port Number
- Proxy Host
- Proxy Password
- Proxy Port
- Proxy User
- Qualify Normalized Names
- Read Ahead
- Redirect URI
- Refresh Dirty Cache
- Refresh Schema
- Refresh Token
- Region
- REST Config File
- REST Sample Path
- Sampling Failure Tolerance
- Schema Map

- Scope
- Secret Key
- Security Token
- Server Port Number
- Server Proxy Host
- Server Proxy Password
- Server Proxy Port
- Server Proxy User
- SQL Engine Mode
- SQL Service
- Statement Call Limit
- Statement Call Limit Behavior
- Table
- Token URI
- Transaction Mode
- Truststore
- Truststore Password
- User
- User Agent String
- Validate Server Certificate
- Web Service Fetch Size
- Web Service Pool Size
- Web Service Retry Count
- Web Service Timeout

Access Key

Attribute

AccessKey (ak)

Purpose

Specifies the access key ID used for authenticating with AWS credentials (`AuthenticationMethod=AWS`).

Valid Values

String

where:

String

is the access key ID for your IAM user or AWS account root user.

Default Value

No default value

Access Token

Attribute

AccessToken (atok)

Purpose

Specifies the access token used to authenticate to REST endpoints with OAuth 2.0 enabled. Typically, this option is configured by the application; however, in some scenarios, you may need to secure a token using external processes. In those instances, you can also use this option to set the access token manually.

Valid Values

String

where:

String

is an access token you have obtained from the authentication service.

Notes

- Access tokens are temporary and must be replaced to maintain the session without interruption. The life of an access token is typically one hour.
- See "OAuth 2.0 authentication" for examples and more information.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Array Normalization Threshold

Attribute

ArrayNormalizationThreshold (art)

Purpose

Specifies the length of arrays (in elements) at which the driver begins to normalize elements in arrays to child tables when generating a flattened relational view (`SchemaFormat=Flatten`). In the flattened relational view, elements in arrays are typically flattened into columns in the parent table. This behavior can create tables with a high-number of columns when encountering large arrays or arrays nested in arrays. To avoid the memory and performance issues associated with handling very large tables, you can limit the size of the parent table using this option.

Valid Values

`0` | *x*

where:

x

is the length of arrays (in elements) at which the driver begins to normalize elements in arrays to child tables when generating a flattened view.

Behavior

If set to `0`, all elements in arrays are flattened as columns to the parent table.

If set to *x*, arrays containing a number of elements equal to or greater than the specified value are normalized to child tables when generating a flattened view. In arrays comprising of fewer elements than the specified amount, the elements are flattened into columns in the parent table.

Notes

- The behavior of this option also applies to nested arrays. Therefore, if the length of a nested array exceeds the value specified for this option, a separate child table will be generated for the elements of the nested array.

Default Value

`12`

Authorization Code

Attribute

OAuthCode (oac)

Purpose

Specifies the temporary authorization code that is exchanged for access tokens when OAuth 2.0 authentication is enabled (`AuthenticationMethod=OAuth2`). Authorization codes are used to authenticate against the endpoint specified by the Token URI option. If authentication is successful, an access token is generated and fetched from the specified location. Typically, this option is configured by the application.

Valid Values

`String`

where:

`String`

is an OAuth 2.0 authorization code.

Notes

- Authorization codes are used to authenticate against the endpoint specified by the Token URI option. If authentication is successful, an access token is generated and fetched from the specified location. Typically, this option is configured by the application.
- See "OAuth 2.0 authentication" for more information.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Authentication Header

Attribute

AuthHeader (ah)

Purpose

Specifies the name of the HTTP header used for authentication. This option is used when Basic (`AuthenticationMethod=0`), Header-based token authentication (`AuthenticationMethod=25`), or OAuth 2.0 authentication is enabled; otherwise, this option is ignored.

Valid Values

`auth_header`

where:

`auth_header`

is the name of the HTTP header used for authentication. For example, `X-Api-Key`.

Default Value

`Authorization`

See also

Authentication on page 92

Authentication Method

Attribute

AuthenticationMethod (am)

Purpose

Determines which authentication method the driver uses during the course of a session.

Valid Values

`15` | `43` | `0` | `34` | `24` | `50` | `51` | `56` | `52` | `53` | `54` | `55` | `25` | `14` | `30` | `29`

Behavior

If set to `15` (None), the driver does not attempt to authenticate.

If set to `43` (AWS), the driver uses AWS (Amazon Web Services) credentials for authentication. You must also configure the AccessKey, Region, and SecretKey options.

If set to `0` (Basic), the driver uses a hashed value, based on the concatenation of the user name and password, for authentication. In addition to the User and Password options, you must also configure the AuthHeader option if the name of your HTTP header is not `Authorization` (the default).

If set to `34` (BearerToken), the driver uses an API Token, configured as BearerToken, for authentication. The BearerToken is specified via the SecurityToken option.

If set to `24` (OAuth2), the driver uses OAuth 2.0 to authenticate to REST endpoints. This is a legacy value that allows you to connect to all supported grant flows. When this value is set, the driver determines which grant type to use based on the OAuth 2.0 related options you specify. This setting differs from other OAuth 2.0 values in that, when specified, it exposes all OAuth 2.0 related options on the Configuration Manager, instead of only those related to a specified grant type. See "OAuth 2.0 authentication" for details.

If set to `50` (OAuth2-AccessToken), the driver uses the access token authentication flow to authenticate to REST endpoints. See "OAuth 2.0 authentication" for details.

If set to `51` (OAuth2-AuthorizationCode), the driver uses authorization code grant to authenticate to REST endpoints. When `EnableLoginPrompt=1` and `SQLEngineMode=2`, the driver uses dynamic authorization flow.

If set to `56` (OAuth2-PKCE), the driver uses the PKCE grant to authenticate to REST endpoints. Note that this option is supported only when the SQL Engine is running in direct mode (`SQLEngineMode=2`) See "OAuth 2.0 authentication" for details.

If set to `52` (OAuth2-Password),the driver uses the password grant authentication to authenticate to REST endpoints. See "OAuth 2.0 authentication" for details.

If set to `53` (OAuth2-ClientCredentials), the driver uses the client credentials grant to authenticate to REST endpoints. See "OAuth 2.0 authentication" for details.

If set to `54` (OAuth2-RefreshToken), the driver uses the refresh token grant to authenticate to REST endpoints. See "OAuth 2.0 authentication" for details.

If set to 55 (OAuth2-JWTBearer), the driver uses the JWT bearer token grant to authenticate to REST endpoints. See "OAuth 2.0 authentication" for details.

If set to 25 (HTTPHeader), the driver passes security tokens via HTTP headers for authentication. You must also configure SecurityToken option and, if the name of your HTTP header is not `Authorization` (the default), the AuthHeader option.

If set to 14 (URLParameter), the driver passes security tokens via the URL for authentication. You must also configure the AuthParam and SecurityToken options.

If set to 30 (Digest), the driver uses digest access authentication to negotiate username and password authentication. You must also configure the User and Password options.

If set to 29 (Custom), the driver uses custom authentication requests specified in the REST config file to login to server and generate the credentials needed to authenticate data requests.

Default Value

15 (None)

See also

Authentication on page 92

Authentication URL Parameter

Attribute

AuthParam (ap)

Purpose

Specifies the name of the URL parameter used to pass the security token. This option is required when using URL parameters to pass tokens for authentication (AuthenticationMethod=UrlParameter); otherwise, this option is ignored

Valid Values

auth_parameter

where:

auth_parameter

is the name of the URL parameter used to pass the security token. For example, `apikey` or `key`.

Behavior

For example, for the URL `https://www.example.com/path?apikey=123`, the parameter name is `apikey`.

Default Value

No default value

See also

Authentication on page 92

Authorization URI

Attribute

AuthURI (o2au)

Purpose

Specifies the endpoint for obtaining an authorization code from a third-party authorization service for OAuth 2.0 implementations.

Valid Values

String

where:

String

is the endpoint for retrieving the OAuth 2.0 authorization code from the third party authorization service.

Notes

- When this endpoint is queried, the authorization service presents an interface prompting the user to approve or deny access to backend data.
- See "OAuth 2.0 authentication" for examples and more information.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Claims Issuer

Attribute

ClaimsIssuer (cliss)

Purpose

Specifies the consumer key or the client ID of the authorization server when authenticating using the OAuth 2.0 JWT bearer grant type. This option must be specified when the JWT bearer grant type is enabled.

Valid Values

String

where:

String

is the consumer key or client ID.

Notes

- See "OAuth 2.0 authentication" for more information.

Default Value

No default value

Claims Subject

Attribute

ClaimsSubject (clsub)

Purpose

Specifies the identifier of the principle that is the subject of the JWT. This option must be specified when authenticating with the OAuth 2.0 JWT bearer grant type.

Valid Values

`String`

where:

`String`

is the identifier of the principle that is the subject of the JWT. The principle can be a user, an organization, or a service.

Notes

- See "OAuth 2.0 authentication" for more information.

Default Value

No default value

Client ID

Attribute

ClientID (cid)

Purpose

Specifies the client ID key for your application when authenticating to REST endpoints with OAuth 2.0 enabled `(AuthenticationMethod=OAuth2).`

Valid Values

String

where:

String

is the client ID key for your application.

Notes

- In some cases, the value for this option is the same as your user name or your authenticating email address. In others, this value is supplied with your client secret. If you experience an authentication error, verify that you are using the correct value.
- See "OAuth 2.0 authentication" for more information.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Client Secret

Attribute

ClientSecret (clse)

Purpose

Specifies the client secret for your application when authenticating to REST endpoints with OAuth 2.0 enabled.

Important: The client secret is a confidential value used to authenticate the application to the service. To prevent unauthorized access, this value must be securely maintained.

Valid Values

String

where:

String

is the client secret for your application.

Notes

See "OAuth 2.0 authentication" for more information.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Create Map

Attribute

CreateMap (cm)

Purpose

Determines whether the driver creates the internal files required for a relational map of the native data model when establishing a connection.

Valid Values

`0` | `1` | `2` | `3`

Behavior

If set to `0` (NotExist), the driver uses the current group of internal files specified by the Schema Map (SchemaMap) option. If the files do not exist, the driver creates them.

If set to `1` (ForceNew), the driver deletes the group of internal files specified by the Schema Map option and creates a new group of these files at the same location.

If set to `2` (No), the driver uses the current group of internal files specified by the Schema Map option. If the files do not exist, the connection fails.

If set to `3` (Session), the driver uses memory to store the internal configuration information and relational map of the native data model. After the session, the view is discarded.

Notes

- The internal files share the same directory as the schema map configuration file. This directory is specified with the Schema Map connection option.

Default Value

`3` (Session)

Crypto Protocol Version

Attribute

CryptoProtocolVersion (cpv)

Purpose

Specifies a cryptographic protocol or comma-separated list of cryptographic protocols that can be used when SSL is enabled.

Valid Values

cryptographic_protocol [[,*cryptographic_protocol*]...]

where:

cryptographic_protocol

is one of the following cryptographic protocols:

TLSv1.3 | TLSv1.2 | TLSv1.1 | TLSv1 | SSLv3 | SSLv2`

Note: The protocols available depend on your Java version. Most modern implementations have disabled all but TLSv1.2 and TLSv1.3.

Caution: To avoid vulnerabilities associated with older protocols, best security practices recommend using TLSv1.2 or higher.

Example

If your server supports TLSv1.2 and TLSv1.3, you can specify acceptable cryptographic protocols with the following key-value pair:

```
CryptoProtocolVersion=TLSv1.2,TLSv1.3
```

Notes

- When multiple protocols are specified, the driver uses the highest version supported by the server. If none of the specified protocols are supported by the server, the connection fails and the driver returns an error.
- The default may be set in the Java system property `https.protocols`, which is often set on the Java command line with the `-D`*`property`*`=` option. For example: `-Dhttps.protocols=TLSv1.2,TLSv1.3`

Default Value

No default value

See also

TLS/SSL encryption on page 121

Custom Authentication Parameters

Attribute

CustomAuthParams (cap)

Purpose

Specifies a list of parameter values used by custom authentication requests that are defined in the Model file. This option allows you to configure parameter values used in custom authentication requests on a per connection basis, without editing the Model file, and securely pass them in a connection string or data source definition.

The Model file references the values of this option using the CustomAuthParams variable followed by an index location surrounded in square brackets. For example, a value of `CustomAuthParams[3]` refers to the third value specified by this option.

Valid Values

authentication_parameter[[;*authentication_parameter*]...]

where:

authentication_parameter

is an authentication parameter value used in a custom authentication requests defined in the Model file. This value can be any parameter value used in the request that is not already mapped to an existing connection option, for example api-key tokens, company names, and website names.

Important: The index value specified for the variable in the Model file corresponds to the order in which these values are specified for the option.

Example

If you needed to reference the value `My Company Inc` for the following company field in the definition for a custom authentication request:

```
"company": "{CustomAuthParams[2]}"
```

Since the variable is pointing to the `2` index location, you would specify `My Company Inc` as the second value in the connection option:

```
CustomAuthParams=123XYZ456abc789;My Company Inc;www.example.com
```

Notes

- This option is enabled when `AuthenticationMethod=Custom`; otherwise, it is ignored.
- The values specified for this option are case insensitive.

Default Value

No default value

See also

Custom authentication on page 119

Data Source Name

Attribute

DataSourceName (dsn)

Purpose

Specifies the name of a data source in your Windows Registry or `odbc.ini` file.

Valid Values

String

where:

String

is the name of a data source.

Default Value

No default value

Debug Folder

Attribute

DebugRecord (dbgrecord)

Purpose

Specifies the directory where the driver generates debug record files. When a value is specified, the driver records server requests and responses to a set of files stored in this location. These files assist in troubleshooting by providing a method for Technical Support to reproduce and debug issues for REST services that are not publicly accessible.

Important: Debug record files may capture security-related headers, such as auth or token headers. Before sending Technical Support debug files, review the content to remove any confidential information that may have been recorded.

Valid Values

`debug_record_folder`

where:

`debug_record_folder`

is the location of the folder where the debug record files are to be generated. For example, `C:\Temp\MyDebug Folder`.

Note: To specify this value using the GUI, you must click the **Record** box . Next, click **Select**; then, browse to or create the file location.

Notes

- The specified directory must exist.
- You must have write access to the specified directory.
- The contents of the specified directory are deleted every time a connection is established.
- For more information, refer to "Enabling debug record mode" in the *Progress DataDirect for ODBC Drivers Reference*.
- For assistance, contact Technical Support.

Default Value

No default value

Default Query Options

Attribute

DefaultQueryOptions (dqo)

Purpose

Specifies a semicolon-separated list of parameters that are used as default filter values (Where clauses) for SQL queries. When querying a table, the default query parameters specified by this option (or filters specified in SQL Where clauses) are appended as query parameters to REST API calls to the endpoint for the queried table.

Valid Values

string

where:

string

is a set of parameters and default filter values that you want to apply to SQL queries.

Examples

If you specified the following value for this option:

```
invoice>=2000;region=americas
```

Or specified the following equivalent value in a SQL statement:

```
SELECT customer_name FROM accounts WHERE invoice >= 2000 AND region = americas
```

The following REST API call would be called to the driver to return results:

```
https://www.example.com/times/query?invoice>=2000&region=americas
```

Notes

- A Where clause used in a SQL query overrides the DefaultQueryOptions passed in a connection URL.

Default Value

No default value

Description

Attribute

Description (desc)

Purpose

Specifies an optional long description of a data source. This description is not used as a runtime connection attribute, but does appear in the `ODBC.INI` section of the Registry and in the `odbc.ini` file.

Valid Values

`String`

where:

`String`

is a description of a data source.

Default Value

No default value

Enable Login Prompt

Attribute

EnableLoginPrompt (elp)

Purpose

Specifies whether the driver fetches access and refresh tokens at connection when OAuth 2.0 Authorization Code Grant is enabled (`AuthenticationMethod=51`). When this option is enabled, the driver launches the login prompt for your service at connection, which allows you to specify your login credentials and initiate the dynamic authorization code grant flow. Enabling this option provides a method of fetching access and refresh tokens without using the Configuration Manager or third-party tool.

Valid Values

`0` | `1`

Behavior

If set to `0` (false), the driver does not launch the login prompt for your service. If access and refresh tokens are needed for your authentication flow, you will need to fetch them using the Configuration Manager or another tool.

If set to `1` (true), the driver opens the login prompt for your service when attempting to connect. Submitting user and password credentials via the prompt initiates the dynamic authorization code grant to fetch access and refresh tokens.

Notes

- This option is used only for the dynamic authorization code grant flow. See "Dynamic authorization code grant" for a full list of requirements.
- This option is supported only when the SQL Engine is running in direct mode (`SQLEngineMode=2`).
- When this option is enabled, the value of the Redirect URI (`RedirectURI`) option must include the port number. For example, `RedirectURI=http://localhost:80` or `RedirectURI=http://localhost:8080`.

Default Value

0 (false)

Encryption Method

Attribute

EncryptionMethod (em)

Purpose

Determines whether data is encrypted and decrypted when transmitted over the network between the driver and REST service.

Valid Values

0 | 1

Behavior

If set to 1 (SSL), data is encrypted using SSL. If the endpoint does not support SSL, the connection fails and the driver throws an exception.

If set to 0 (NoEncryption), data is not encrypted or decrypted.

Default Value

- SSL encryption is enabled when the URL specified in the REST Sample Path (RESTSamplePath) option or Model file uses HTTPS, regardless of the setting of Encryption Method.
- When SSL is enabled, the following options also apply:
  - Crypto Protocol Version
  - Host Name In Certificate
  - Key Password (for SSL client authentication)
  - KeyStore (for SSL client authentication)
  - KeyStore Password (for SSL client authentication)
  - TrustStore
  - TrustStore Password
  - Validate Server Certificate

See also

REST Sample Path on page 180
TLS/SSL encryption on page 121

Extended Options

Attribute

ExtendedOptions (xo)

Purpose

Specifies a semicolon separated list of connection options and their values. Use this connection option to set the value of undocumented connection options that are provided by Progress DataDirect Technical Support.

Valid Values

option=*value*[;*option*=*value*;...]

where:

option

is a connection option.

value

is the value or setting of the connection option.

Default Value

No default value

Fetch Size

Attribute

FetchSize (fs)

Purpose

Specifies the maximum number of rows that the driver processes before returning data to the application when executing a Select. This value provides a suggestion to the driver as to the number of rows it should internally process before returning control to the application. The driver may fetch fewer rows to conserve memory when processing exceptionally wide rows.

Valid Values

0 | *x*

where:

x

is a positive integer indicating the number of rows that should be processed.

Behavior

If set to `0`, the driver processes all the rows of the result before returning control to the application. When large data sets are being processed, setting Fetch Size to `0` can diminish performance and increase the likelihood of out-of-memory errors.

If set to *x*, the driver limits the number of rows that may be processed for each fetch request before returning control to the application.

Notes

- To optimize throughput and conserve memory, the driver uses an internal algorithm to determine how many rows should be processed based on the width of rows in the result set. Therefore, the driver may process fewer rows than specified by Fetch Size when the result set contains exceptionally wide rows. Alternatively, the driver processes the number of rows specified by Fetch Size when the result set contains rows of unexceptional width.
- You can use Fetch Size to reduce demands on memory and decrease the likelihood of out-of-memory errors. Simply, decrease Fetch Size to reduce the number of rows the driver is required to process before returning data to the application.

Default Value

`100`

See also

Performance considerations on page 124

Health Check URI

Attribute

HealthURI (huri)

Purpose

Specifies the URI that the driver calls to confirm connectivity when using certain authentication methods.

Services using some authentication methods, such as Basic, Digest, URL Parameter-based, or HTTP header-based, do not perform an explicit action upon connection. Instead, authentication occurs only when query or update operations are executed. As a result, the driver does not receive feedback to determine whether a connection attempt was successful. You can work around this limitation by specifying a value for this option. At connection, when a value is specified, the driver issues a request to the specified URI, analyzes the result status, and then discards the result.

Valid Values

test_uri

where:

test_uri

is the absolute or relative URI against which the driver executes a query to test connectivity. For optimal performance, this value should be an endpoint that executes quickly and has a small response.

Notes

- The driver performs a connectivity test at connection, when executing a test connect, or whenever the driver needs to confirm connectivity.
- If no value is specified for this option, executing a test connect on the Configuration Manager will return a message that the connection was successful every time.

Default Value

No default value

See also

Authentication on page 92

Host Name

Attribute

HostName (host)

Purpose

Specifies the host name portion of the HTTP endpoint to which you send requests.

Valid Values

url

where:

url

is the host name portion of the HTTP endpoint to which you send requests.

Notes

- The ServerName attribute is an alias for the Host Name (HostName) option.

Default Value

No default value

Host Name In Certificate

Attribute

HostNameInCertificate (hnic)

Purpose

Specifies a host name for certificate validation when SSL encryption is enabled and validation is enabled (`ValidateServerCertificate=1`). This option is optional and provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

Valid Values

`host_name`

where:

`host_name`

is a valid host name.

Behavior

If *`host_name`* is specified, the driver compares the specified host name to the DNSName value of the SubjectAlternativeName in the certificate. If the certificate does not have a SubjectAlternativeName, the driver compares the host name with the Common Name (CN) part of the certificate. If the values do not match, the connection fails and the driver throws an exception.

Notes

- If SSL encryption or certificate validation is not enabled, this option is ignored.
- If SSL encryption and validation is enabled and this option is unspecified, the driver uses the value of the Host Name (HostName) option to validate the certificate.

Default Value

No default value

See also

TLS/SSL encryption on page 121

JSON Root

Attribute

JSONRoot (jr)

Purpose

Specifies the embedded object, including the path, that contains the results you want mapped to a dedicated relational table. When specifying an endpoint using the REST Sample Path (RESTSamplePath) option method, this option allows you to limit the results mapped to the table to only those returned in the specified object, instead of the results from the entire endpoint.

Valid Values

`String`

where

String is the embedded object and path that contains the results you want mapped to a dedicated relational table. For nested objects, separate the element names with forward slashes.

Example

For example, an endpoint returns the following JSON response that contains multiple objects, `total_events` and `events`.

```
{
  "total_events": 2,
  "events": [
    {
      "attending_count": 164,
      "cost": 6899.00,
      "name": "Smith Family Reunion",
      "location": "San Francisco"
    },
    {
      "attending_count": 87,
      "cost": 5465.00,
      "name": "Brady Wedding",
      "location": "Santa Clara"
    }
  ]
}
```

If no value is specified for this option, the driver maps the `total_events` object to a parent table and `events` object to a child table. However, if your application only needs the information from the `events` object, you can specify `/events` with this option to map only that object. In this configuration, the driver maps the `events` object to a parent table and each element of the array to rows, while the `total_events` object is ignored.

Notes

- This option is used only when specifying an endpoint using the REST Sample Path (RESTSamplePath) option method. When using a Model file, you can use the JSON Root field on the Autonomous REST Composer to specify this value. See "Generating a Model file with the Autonomous REST Composer" for more information.
- During sampling, the driver specifies a default value for this option when it detects an embedded object that meets the criteria for a JSON root. Depending on the structure of your data model, you might want to add, edit, or remove values for this option.

Default Value

No default value

JVM Arguments

Attribute

JVMArgs (jvma)

Purpose

A string that contains the arguments that are passed to the JVM that the driver is starting. The location of the JVM must be specified on the driver library path. For information on setting the location of the JVM in your environment, see:

- "Windows environment variables"
- "Library search path" (Linux)

When specifying the heap size for the JVM, the JVM tries to allocate the heap memory as a single contiguous range of addresses in the application's memory address space. If the application's address space is fragmented so that there is no contiguous range of addresses big enough for the amount of memory specified for the JVM, the driver fails to load, because the JVM cannot allocate its heap. This situation is typically encountered only with 32-bit applications, which have a much smaller application address space. If you encounter problems with loading the driver in an application, try reducing the amount of memory requested for the JVM heap. If possible, switch to a 64-bit version of the application.

Valid Values

String

where:

String

contains arguments that are defined by the JVM. Values that include special characters or spaces must be enclosed in curly braces { } when used in a connection string.

Example

To set the heap size used by the JVM to 256 MB and the http proxy information, specify:

```
{-Xmx256m -Dhttp.proxyHost=johndoe -Dhttp.proxyPort=808}
```

To set the heap size to 256 MB and configure the JVM for remote debugging, specify:

```
{-Xmx256m -Xrunjdwp:transport=dt_socket, address=9003,server=y,suspend=n -Xdebug}
```

Default Value

For the 32-bit driver when the SQL Engine Mode connection option is set to 2 (Direct):

`-Xmx256m`

For all other configurations:

`-Xmx1024m`

See also

Using the SQL engine server on page 125

JVM Classpath

Attribute

JVMClassPath (jvmcp)

Purpose

Specifies the CLASSPATH for the Java Virtual Machine (JVM) used by the driver. The CLASSPATH is the search string the JVM uses to locate the Java jar files the driver needs.

Valid Values

`String`

where:

`String`

specifies the CLASSPATH. Separate multiple jar files by a semi-colon on Windows platforms and by a colon on Linux platforms. CLASSPATH values with multiple jar files must be enclosed in curly braces `{ }` when used in a connection string.

If your process employs multiple drivers that use a JVM, the value of the JVM Classpath for all affected drivers must include an absolute path to all the jar files for drivers used in that process. In addition, the value specified must be identical for all drivers. For example, if you are using the Autonomous REST Connector and MongoDB driver on Windows, you would specify a value of `{c:\install_dir\java\lib\autorest.jar; c:\install_dir\java\lib\mongodb.jar}` for both drivers. If the value for any of the affected drivers is missing a file path or is different from the one specified for the other drivers, the drivers will return an error at connection that the JVM is already running.

Example

On Windows:

`{.;c:\install_dir\java\lib\autorest.jar}`

On Linux:

`{.:/home/user1/install_dir/java/lib/autorest.jar}`

Default Value

No default value

See also

Using the SQL engine server on page 125

JVM Path

Attribute

JVMPath (jvmp)

Purpose

Specifies fully qualified path to the JVM executable that you want to use to run the SQL Engine Server. The path must not contain double quotation marks.

Valid Values

`String`

where:

`String`

The full path to the JVM executable.

Default Value

install_dir\jre\bin\java.exe

See also

Using the SQL engine server on page 125

JWT Certificate Alias

Attribute

JWTCertAlias (jwtca)

Purpose

Specifies the alias for the JWT certificate stored in the keystore when authenticating using the OAuth 2.0 JWT bearer grant type. This option is optional.

Valid Values

String

where:

String

is the alias for the JWT certificate.

Notes

- See "OAuth 2.0 authentication" for more information.

Default Value

No default value

JWT Certificate Password

Attribute

JWTCertPassword (jwtcp)

Purpose

Specifies the password of the JWT certificate when authenticating using the OAuth 2.0 JWT bearer grant type. A value for this option is specified only if the certificate is password protected.

Valid Values

String

where:

String

is the password for the certificate.

Notes

- See "OAuth 2.0 authentication" for more information.

Default Value

No default value

JWT Certificate Store

Attribute

JWTCertStore (jwtcs)

Purpose

Specifies the file path of the certificate store containing the private key used for the OAuth 2.0 JWT bearer grant type. This keystore can be a JKS file or a PKCS12 file. This option must be specified when JWT Bearer grant authentication is enabled.

Valid Values

`String`

where:

`String`

is the file path for the JWT certificate store.

Notes

- See "OAuth 2.0 authentication" for more information.

Default Value

No default value

Key Password

Attribute

KeyPassword (kp)

Purpose

Specifies the password that is used to access the individual keys in the keystore file when SSL is enabled and SSL client authentication is enabled on the server. This property is useful when individual keys in the keystore file have a different password than the keystore file.

Valid Values

key_password

where:

key_password

is a valid password.

Default Value

No default value

See also

TLS/SSL encryption on page 121

Keystore

Attribute

Keystore (ks)

Purpose

Specifies the directory of the keystore file to be used when SSL is enabled and SSL client authentication is enabled on the server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.

Valid Values

keystore_directory

where:

keystore_directory

is a valid directory of a keystore file.

Notes

- This value overrides the directory of the keystore file that is specified by the `javax.net.ssl.keyStore` Java system property. If this property is not specified, the keystore directory is specified by the `javax.net.ssl.keyStore` Java system property.
- The keystore and truststore files can be the same file.

Default Value

No default value

Keystore Password

Attribute

KeystorePassword (ksp)

Purpose

Specifies the password that is used to access the keystore file when SSL is enabled and SSL client authentication is enabled on the server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.

Valid Values

string

where:

string

is a valid password.

Notes

- This value overrides the password of the keystore file that is specified by the `javax.net.ssl.keyStorePassword` Java system property. If this property is not specified, the keystore password is specified by the `javax.net.ssl.keyStorePassword` Java system property.
- The keystore and truststore files can be the same file; therefore, they may have the same password.

Default Value

No default value

See also

TLS/SSL encryption on page 121

Log Config File

Attribute

LogConfigFile (lgcf)

Purpose

Specifies the file name, and optionally, the path of the properties file used to initialize driver logging.

Valid Values

String

where:

String

is the relative or fully qualified path of the properties file to load to initialize driver logging. If you do not specify a path, the driver looks for this file in the current working directory. If the specified file does not exist, the driver continues searching for an appropriate properties file as described in "Logging for Java components" in the *Progress DataDirect for ODBC Drivers Reference*.

Default Value

`ddlogging.properties`

Logoff URI

Attribute

LogoffURI (o2lu)

Purpose

Specifies the endpoint the driver calls to notify the service to log the client out of the session, including performing any clean-up tasks or expiring the token. The driver uses this value when authenticating to a REST service using OAuth 2.0.

Valid Values

`string`

where:

`string`

is the endpoint used to retrieve OAuth 2.0 authorization codes. For example:
`https://example.com/oauth2/logout/`

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

OAuth Client Credentials Mode

Attribute

ClientCredentialsMode (clcm)

Purpose

Determines how client credentials are sent in a request to obtain an access token when using OAuth 2.0. Configure this option for flows that require client credentials to be specified as only a basic authentication header or as only a URL parameter.

Valid Values

`0` | `1` | `2` | `3`

Behavior

If set to `0` (Default), the client credentials are sent as a basic authentication header.

If set to `1` (Basic), the client credentials are sent as a basic authentication header.

If set to `2` (URL), the client credentials are sent as a URL parameter.

If set to `3` (Post), the client credentials are sent in the body of a POST request.

Notes

- This option is not required for all authentication flows. If you are unsure of the requirements for your authentication flow, contact your administrator for more information.

Default Value

`0` (Default)

Password

Attribute

Password (pwd)

Purpose

A password that is used to connect to the service.

Behavior

`password`

where:

`password`

is a valid password. The password is case-sensitive.

Default Value

No default value

See also

Authentication on page 92

Port Number

Attribute

PortNumber (port)

Purpose

Specifies the TCP port of the server that is listening for REST API requests.

Valid Values

port_number

where:

port_number

is the port number.

Default Value

When SSL encryption is disabled:

`80`

When SSL encryption is enabled:

`443`

Proxy Host

Attribute

ProxyHost (pxhn)

Purpose

Identifies a proxy server to use for the first connection.

Valid Values

server_name | *IP_address*

where:

server_name

is the name of the proxy server, which may be qualified with the domain name.

IP_address

is an IP address, specified in either IPv4 or IPv6 format, or a combination of the two.

Default Value

No default value

See also

Connecting through a proxy server on page 123

Proxy Password

Attribute

ProxyPassword (pxpw)

Purpose

Specifies the password needed to connect to a proxy server for the first connection.

Valid Values

password

where:

password

is a valid password for that server. Contact your system administrator to obtain a valid password.

Default Value

No default value

See also

Connecting through a proxy server on page 123

Proxy Port

Attribute

ProxyPort (pxpt)

Purpose

Specifies the port number where the proxy server is listening for HTTP or HTTPS requests for the first connection.

Valid Values

port

where:

port

is the port number on which the proxy server is listening. Contact your system administrator to obtain the correct port.

Default Value

`0` which means that the default value is determined by whether the value specified for the Proxy Host (ProxyHost) option is an HTTP or HTTPS URL.

For HTTP: `80`

For HTTPS: `443`

See also

Connecting through a proxy server on page 123

Proxy User

Attribute

ProxyUser (pxun)

Purpose

Specifies the user name needed to connect to a proxy server for the first connection.

Valid Values

user_name

where:

user_name

is a valid user ID for the proxy server.

Default Value

No default value

See also

Connecting through a proxy server on page 123

Qualify Normalized Names

Attribute

QualifyNormalizedNames (qnn)

Purpose

Determines whether the names of relational tables normalized from array columns are derived directly from the column name or prefixed with parent object names.

Valid Values

`0` | `1` | `2`

Behavior

If set to 0 (No), the relational table name is derived solely from the column name of the array.

If set to 1 (Table), the relational table name is prepended with the name of the immediate parent object. For example, if the immediate parent was Books and the array column was Chapters, then the relational table name would be BOOKS_CHAPTERS.

If set to 2 (FullPath), the relational table name is prepended with the names of all objects in which the array column is nested. For example, if the immediate parent was BOOKS with an array Chapters that contained an array Pages, then the resulting relational table name would be BOOKS_CHAPTERS_PAGES.

Notes

- If a naming conflict occurs, the driver appends an underscore separator and integer (for example, _1) to the table name.
- The value of this option also controls the name of the foreign key column in the child table. For example, if this option is set to Table, the foreign key column name would be _ID prepended with the parent object name. Therefore, a parent object of BOOKS would result in a foreign key column named BOOKS_ID in the child table.

Default Value

0 (No)

See also

Mapping objects to tables on page 40

Read Ahead

Attribute

ReadAhead (ra)

Purpose

Specifies the maximum number of fetch requests the driver issues in parallel. By default, the driver queues the next page when processing the current page. This option allows you to fetch multiple requests simultaneously, thereby improving throughput and performance.

Caution: Due to potential impacts to other users on the network, we strongly recommend specifying only smaller values for this option. For example, in fully optimized environments, which include exceptionally fast connections and low latency, we recommend a setting of no higher than 10. For typical environments, this value should be considerably lower.

Valid Values

0 | *x*

where:

x

is the maximum number of fetch requests the driver issues in parallel up to 100.

Behavior

If set to `0`, the driver queues the next page while processing the current page.

If set to *x*, the driver executes fetch requests as they are issued until the number of active parallel-requests equals the specified value. When that threshold is met, the driver waits until the results of a request are processed before requesting the next page of data.

Notes

- Specifying larger values for this option generally improves performance; however, with the following warnings:
  - Larger values can increase the load on the server, which may adversely affect performance of other users. If you encounter issues, decrease the value specified for this option.
  - Larger values may result in unnecessary requests if your application only requires the first few rows of results. This may be an issue if your service places limits on the number of web requests.

Default Value

`0`

See also

Performance considerations on page 124

Redirect URI

Attribute

RedirectURI (o2ru)

Purpose

Specifies the endpoint to which the client is returned after authenticating with a third-party service when OAuth 2.0 authentication is enabled `(AuthenticationMethod=OAuth2)`.

For some authentication flows, the REST service will redirect you to a third-party service for authentication. Once your credentials have been validated, the third-party service returns the client to an endpoint in the REST service to continue the session. The endpoint used by the third-party service is provided by the client and specified using the Redirect URI option.

Valid Values

`String`

where:

`String`

is the endpoint used to retrieve OAuth 2.0 authorization codes. For example, `https://example.com/countries/`.

Notes

- The redirect endpoint is often registered with the authentication service to provide improved security. Registering the endpoint prevents your valid authentication credentials being redirected to a malicious site; therefore, reducing the risk of sharing your access token and other sensitive information with unauthorized parties.
- See "OAuth 2.0 authentication" for examples and more information.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Refresh Dirty Cache

Attribute

RefreshDirtyCache (rdc)

Purpose

Specifies whether the driver refreshes a dirty cache on the next fetch operation from the cache. A cache is marked as dirty when a row is inserted into or deleted from a cached table or a row in the cached table is updated.

Valid Values

`0` | `1`

Behavior

If set to `1` (true), a dirty cache is refreshed when the cache is referenced in a fetch operation. The cache state is set to initialized if the refresh succeeds.

If set to `0` (false), a dirty cache is not refreshed when the cache is referenced in a fetch operation.

Default Value

`1` (true)

Refresh Schema

Attribute

RefreshSchema (rs)

Purpose

Specifies whether the driver automatically refreshes the relational map of the schema when a user connects to the service.

Valid Values

`0` | `1`

Behavior

If set to `1` (true), the driver automatically refreshes the map when a user first connects to the service. The driver rebuilds the relational map of the schema, and any changes that have been made to the schema since the last refresh will be shown in the metadata.

If set to `0` (false), the driver does not refresh the relational map when a user first connects.

Notes

- This option should not be enabled (`RefreshSchema=1`) when `CreateMap=3` (Session).
- This option is equivalent to executing the Refresh Map statement.

Default Value

`0` (false)

Refresh Token

Attribute

RefreshToken (rtok)

Purpose

Specifies the refresh token used to either request a new access token or renew an expired access token for OAuth 2.0 implementations.

Important: The refresh token is a confidential value used to authenticate to the service. To prevent unauthorized access, this value must be securely maintained.

Valid Values

`String`

where:

`String`

is the refresh token you have obtained from the authentication service.

Notes

- See "OAuth 2.0 authentication" for more information.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Region

Attribute

Region

Purpose

Specifies the name of the region that hosts your AWS server when using AWS credentials to authenticate (`AuthenticationMethod=AWS`).

Valid Values

`String`

where:

`String`

is the name of the region that hosts your AWS server. For example, `us-east-1` or `us-east-2`.

Default Value

No default value

REST Config File

Attribute

RESTConfigFile (rcf)

Purpose

Specifies the name and location of the Model file used to define your endpoints for sampling. This file allows you to specify multiple endpoints, define POST requests, and configure paging. You will need to create and specify a Model file if your session:

- Accesses multiple endpoints
- Issues POST requests
- Accesses endpoints that require paging
- Accesses endpoints that use custom HTTP-headers
- Uses custom HTTP response code processing
- Requires a custom authentication flow

Valid Values

String

where:

String

is the name and location of your Model file. For example, `C:\path\to\myrest.rest` (Windows) or `<home_dir>/path/to/myrest.rest` (UNIX/Linux).

Notes

- In most scenarios, you will configure the REST Config File option, for specifying a Model File, or the REST Sample Path option, for specifying a single endpoint to sample. However, note that these options are not required for all use cases, such as when the driver is being used to execute dynamically created stored procedures or functions.

Default Value

No default value

See also

Generating a Model file with the Autonomous REST Composer on page 70

REST Sample Path

Attribute

RESTSamplePath (rsp)

Purpose

Specifies the endpoint that the driver connects to and samples. This option allows you to configure the driver to issue GET requests to a single endpoint without creating a Model file. Note that if your session does any of the following, instead of using this option, you must create a Model file and specify its location with the REST Config File (RESTConfigFile) option:

- Accesses multiple endpoints
- Issues POST requests
- Accesses endpoints that require paging
- Accesses endpoints that use custom HTTP headers
- Uses custom HTTP response code processing
- Requires a custom authentication flow

Valid Values

URI

where:

URI

is the endpoint that you connect to and sample. For example, `https://example.com/countries/`. Note that the value must be valid URL-encoded syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.

Notes

- Specifying an HTTPS endpoint using the REST Sample Path option enables SSL data encryption. See "Data encryption options" for a list of related options.
- When using the REST Sample Path option, the driver maps endpoints that consist of only a host name to the `URL_` parent table by default.
- In most scenarios, you will configure the REST Config File option, for specifying a Model File, or the REST Sample Path option, for specifying a single endpoint to sample. However, note that these options are not required for all use cases, such as when the driver is being used to execute dynamically created stored procedures or functions.

Default Value

No default value

See also

TLS/SSL encryption on page 121
URL-encoded values on page 264

Sampling Failure Tolerance

Attribute

SamplingFailureTolerance (sft)

Purpose

Specifies the number of endpoints for which sampling can fail before the driver fails the connection.

Valid Values

`-1` | *`x`*

where:

`x`

is the number of endpoints for which sampling can fail before the driver fails the connection. This value can be an integer from `0` to `1024`.

Behavior

If set to `-1`, the driver only fails the connection if sampling for every endpoint in the Model fails. If the connection fails, the driver returns an error message.

If set to *`x`*, the driver fails the connection and returns an error message if the number of endpoints in the model that are unable to be sampled exceeds the specified number.

Notes

You can troubleshoot endpoints in your Model by querying the `INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS` table. This table provides sampling status information for all the endpoints that you specified in your Model. If your connection is failing, we recommend that you set `SamplingFailureTolerance=-1` to ensure status from all your endpoints are included in the table.

Default Value

`-1`

See also

Reviewing the status of your endpoints on page 80

Schema Map

Attribute

SchemaMap (smp)

Purpose

Specifies the directory where the internal configuration files, REST file, and the relational map of the REST data model are written. The driver looks for these files when connecting to a REST service. If the file does not exist, the driver creates one.

Valid Values

`string`

where:

`string`

is the path to the directory used to store the configuration files, REST file, and relational map. For example, if SchemaMap is set to a value of `C:\\Users\\Default\\AppData\\Local\\Progress\\DataDirect\\AutoREST_Schema\\`, the driver either creates or looks for these files in the directory `C:\Users\Default\AppData\Local\Progress\DataDirect\AutoREST_Schema`.

Notes

- By default, the name of the internal files are determined by the values of the User (User) and Host Name (HostName) connection options. If no value is specified for either option, a prefix of USER is used for these files. For example, `USER.config`.
- When connecting to a REST service, the driver looks for the schema map configuration files in the specified location. If the configuration files do not exist, the driver creates them using the location you have provided. If you do not provide a location, the driver creates it using default values.

Default Value

- For Windows platforms
  - User data source: `<user_profile>\AppData\Local\Progress\DataDirect\AutoREST_Schema\`

    - System data source:
      `C:\Users\Default\AppData\Local\Progress\DataDirect\AutoREST_Schema\`

- For UNIX/Linux

    - `<home_dir>/progress/datadirect/autorest_schema/`

Scope

Attribute

Scope (oas)

Purpose

Specifies a space-separated list of OAuth scopes that limit the permissions granted by an access token.

Valid Values

String

where:

String

is a space-separated list of security scopes.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Secret Key

Attribute

SecretKey (sk)

Purpose

Specifies the secret access key used for authenticating with AWS credentials (`AuthenticationMethod=AWS`).

Important: The secret access key is a confidential value used to authenticate the application to the service. To prevent unauthorized access, this value must be securely maintained.

Valid Values

String

where:

String

is the secret access key for your IAM user or AWS account root user.

Default Value

No default value

Security Token

Attribute

SecurityToken (st)

Purpose

Specifies the security token or other HTTP header value used to make a connection to the server.

This option is required when token-based authentication is enabled (`AuthenticationMethod=34` [Bearer Token] | `25` [HttpHeader] | `14` [UrlParameter]). If a security token is required and you do not supply one, the driver returns an error indicating that an invalid user or password was supplied.

For OAuth 2.0 flows, this option specifies the value of custom HTTP headers named by the Authentication Header (AuthHeader) option. If a value for the Authentication Header option is specified, and you do not supply a value for this option, the driver returns an error. This functionality is often used to pass the session string for tenant ID authentication.

Important: If setting the security token using an `odbc.ini` or `odbcinst.ini` files, be aware that the value of this option, like all options in these files, is persisted in clear text.

Valid Values

String

where:

String

is the value of the security token assigned to the user or the value of the custom header to be passed for authentication.

Notes

- When setting the security token using a data source on Windows, the Security Token option is encrypted.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Server Port Number

Attribute

ServerPortNumber (sport)

Purpose

Specifies a valid port on which the SQL engine listens for requests from the driver.

Valid Values

port_number

where:

port_number

is the port number of the server listener. Check with your system administrator for the correct number.

Notes

- This option is ignored when SQL Engine Mode (SQLEngineMode) is set to `2` (Direct).

Default Value

For the 32-bit driver:

`19942`

For the 64-bit driver:

`19941`

See also

Using the SQL engine server on page 125

Server Proxy Host

Attribute

ServerProxyHost (sph)

Purpose

Specifies the host name and possibly the domain of the proxy server used by the SQL engine server. The value specified can be a host name, a fully qualified domain name, or an IPv4 or IPv6 address.

Valid Values

server_name | *ip_address*

where:

server_name

is the name of the proxy server or a fully qualified domain name to which you want to connect.

IP_address

is the IP address of the server. The IP address can be specified in either IPv4 or IPv6 format, or a combination of the two.

Default Value

No default value

See also

Using the SQL engine server on page 125

Server Proxy Password

Attribute

ServerProxyPassword (spw)

Purpose

Specifies the password needed to connect to the proxy server used by the SQL engine server.

Valid Values

string

where:

string

specifies the password to use to connect to the Proxy Server. Contact your system administrator to obtain your password.

Default Value

No default value

Server Proxy Port

Attribute

ServerProxyPort (spp)

Purpose

Specifies the port number of the server listener for the proxy server used by the SQL engine server.

Valid Values

port_name

where:

port_name

is the port number of the server listener of the proxy server used by the SQL engine server. Check with your system administrator for the correct number.

Default Value

No default value

Server Proxy User

Attribute

ServerProxyUser (spu)

Purpose

Specifies the user name needed to connect to the proxy server used by the SQL engine server.

Valid Values

string

where:

string

Is the default user ID that is used to connect to the proxy server used by the SQL engine server.

Default Value

No default value

See also

Using the SQL engine server on page 125

SQL Engine Mode

Attribute

SQLEngineMode (sem)

Purpose

Specifies whether the driver's SQL engine runs in the same process as the driver (direct mode) or runs in a process that is separate from the driver (server mode). You must be an administrator to modify the server mode configuration values, and to start or stop the SQL engine service.

Valid Values

`0` | `1` | `2`

Behavior

If set to `0` (Auto), the SQL engine attempts to run in server mode first; however, if server mode is unavailable, it runs in direct mode. To use server mode with this value, you must start the SQL Engine service before using the driver (see "Using the SQL engine server" for more information).

If set to `1` (Server), the SQL engine runs in server mode. The SQL engine operates in a separate process from the driver within its own JVM. You must start the SQL Engine service before using the driver (see "Using the SQL engine server" for more information).

If set to `2` (Direct), the SQL engine runs in direct mode. The driver and its SQL engine run in a single process within the same JVM.

Important: Changes you make to the server mode configuration affect all DSNs sharing the service.

Default Value

For Windows:

`1` (Server)

For Linux:

`2` (Direct)

See also

Using the SQL engine server on page 125

SQL Service

Attribute

SQLService (ss)

Purpose

Displays the name of the ODBC SQL engine service that runs as a separate process instead of being loaded within the process of an ODBC application.

Note: This option is used only for display purposes in the configuration manager. No value should be specified for this option.

Default Value

No default value

Statement Call Limit

Attribute

StmtCallLimit (scl)

Purpose

Specifies the maximum number of web service calls the driver can make when executing any single SQL statement or metadata query.

Valid Values

`0` | *`x`*

where:

`x` is a positive integer that defines the maximum number of web service calls up to 2147483647 the driver can make when executing any single SQL statement or metadata query.

Behavior

If set to `0`, there is no limit.

If set to *`x`*, the driver uses this value to set the maximum number of web service calls on a single connection that can be made when executing a SQL statement. This limit can be overridden by changing the `STMT_CALL_LIMIT` session attribute using the `ALTER SESSION` statement. For example, the following statement sets the statement call limit to 10 web service calls:

```
ALTER SESSION SET STMT_CALL_LIMIT=10
```

If the web service call limit is exceeded, the behavior of the driver depends on the value specified for the Statement Call Limit Behavior (StmtCallLimitBehavior) option.

Default Value

`0`

See also

Statement Call Limit Behavior on page 189

Statement Call Limit Behavior

Attribute

StmtCallLimitBehavior (sclb)

Purpose

Specifies the behavior of the driver when the maximum web service call limit specified by the Statement Call Limit (StmtCallLimit) option is exceeded.

Valid Values

`0` | `1`

Behavior

If set to `0` (ReturnResults), the driver returns any partial results it received prior to the call limit being exceeded. The driver generates a warning that not all of the results were fetched.

If set to `1` (ErrorAlways), the driver generates an exception if the maximum web service call limit is exceeded.

Default Value

`1` (ErrorAlways)

See also

Statement Call Limit on page 188

Table

Attribute

Table (tbl)

Purpose

Determines the name of the table your endpoint maps to when specifying an endpoint using the REST Sample Path (RESTSamplePath) option. If the table already exists, including those defined in a Model file, the driver will resample the endpoint associated with this table and add any newly discovered columns to the relational view.

Valid Values

`string`

where:

`string`

is the name of the table you want to create for the endpoint or resample.

Notes

- When resampling an existing table, the driver will not remove any columns associated with data that is no longer discoverable in the native view.

Default Value

No default value

See also

REST Sample Path on page 180

Token URI

Attribute

TokenURI (o2tu)

Purpose

Specifies the endpoint for retrieving access tokens when OAuth 2.0 authentication is enabled.

Valid Values

String

where:

String

is the endpoint used to retrieve access tokens.

Notes

- By default, the connector prefixes the token URI endpoint with a GET request method. However, some OAuth implementations require that the token URI endpoint be passed with a POST request method. In this scenario, the token URI endpoint must be prefixed with POST when specifying the value of the TokenURI option. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.
- See "OAuth 2.0 authentication" for more information.

Default Value

No default value

See also

OAuth 2.0 authentication on page 99

Transaction Mode

Attribute

TransactionMode (tm)

Purpose

Specifies how the driver handles manual transactions.

Valid Values

`0` | `1` | `3`

Behavior

If set to `0` (NoTransactions), the data source and the driver do not support transactions. Metadata indicates that the driver does not support transactions.

If set to `1` (Ignore), the data source does not support transactions and the driver always operates in auto-commit mode. Calls to set the driver to manual commit mode and to commit transactions are ignored. Calls to rollback a transaction cause the driver to throw an exception indicating that no transaction is started. Metadata indicates that the driver supports transactions and the ReadUncommitted transaction isolation level.

If set to `3` (Full), the driver passes explicit transaction control commands such as `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` to the service.

Default Value

`3` (Full)

Truststore

Attribute

Truststore (ts)

Purpose

Specifies the directory of the truststore file to be used when SSL is enabled and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

Valid Values

string

where:

string

is the directory of the truststore file.

Notes

- This value overrides the directory of the truststore file that is specified by the javax.net.ssl.trustStore Java system property. If this property is not specified, the truststore directory is specified by the javax.net.ssl.trustStore java system property.
- This option is ignored if ValidServerCertificate=`0`.

Default Value

No default value

See also

TLS/SSL encryption on page 121

Truststore Password

Attribute

TruststorePassword (tsp)

Valid Values

string

where:

string

is a valid password for the truststore file.

Behavior

Specifies the password that is used to access the truststore file when SSL is enabled and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

Notes

- This value overrides the directory of the truststore file that is specified by the javax.net.ssl.trustStore Java system property. If this property is not specified, the truststore directory is specified by the javax.net.ssl.trustStore java system property.
- This option is ignored if ValidServerCertificate=`0`.

Default Value

No default value

See also

TLS/SSL encryption on page 121

User

Attribute

User

Purpose

Specifies the user name that is used to connect to the service.

Valid Values

`String`

where:

`String`

is a valid user name. The user name is case-insensitive.

Default Value

No default value

See also

Authentication on page 92

User Agent String

Attribute

UserAgent (ua)

Purpose

Specifies the string value of the User-Agent header to be used in HTTP requests. This option provides a method to override the default value of the User-Agent header when required by a service.

Note that the default value of the User-Agent header is `Progress Software` *`driver_name`* `Driver`; although, certain internal conditions can cause this value to be altered.

Valid Values

`String`

where:

`String`

is the string value of the User-Agent header.

Example

Some services require the string `gzip` to be in the user agent header to enable GZip compression in their responses, in addition to the usual Accept-Encoding header. For example:

```
UserAgent=Our Stupendous Application (gzip)
```

Other services require contain information, such as an email address, to be included in the user agent header for diagnostic purposes. For example:

```
UserAgent=gcollison@example.com
```

Default Value

No default value

Validate Server Certificate

Attribute

ValidateServerCertificate (vsc)

Purpose

Determines whether the driver validates the certificate that is sent by the database server when accessing an HTTPS endpoint. When using SSL server authentication, any certificate sent by the server must be issued by a trusted Certificate Authority (CA). Allowing the driver to trust any certificate returned from the server even if the issuer is not a trusted CA is useful in test environments because it eliminates the need to specify truststore information on each client in the test environment.

Valid Values

`0` | `1`

If set to `False` (Disabled), the driver does not validate the certificate that is sent by the database server. The driver ignores any truststore information specified by the Truststore and Truststore Password options.

If set to `True` (Enabled), the driver validates the certificate that is sent by the database server. Any certificate from the server must be issued by a trusted CA in the truststore file. If the Host Name In Certificate option is specified, the driver also validates the certificate using a host name. The Host Name In Certificate option provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

Notes

- Truststore information is specified using the TrustStore and TrustStorePassword properties or by using Java system properties.

Default Value

`1` (true)

See also

TLS/SSL encryption on page 121

Web Service Fetch Size

Attribute

WSFetchSize (wsfs)

Purpose

Specifies the number of rows of data the driver attempts to fetch for each JDBC call when paging is enabled for an endpoint.

Note: To enable paging, specify paging parameters for an endpoint in the Model file. See "Creating a Model file" for details.

Valid Values

`0` | *`x`*

where:

`x`

is a positive integer that defines a number of rows. The maximum is defined by the setting of the maximumPageSize property in the Model file.

Behavior

If set to `0`, the driver attempts to fetch up to the maximum number of rows. This value typically provides the maximum throughput.

If set to *`x`*, the driver attempts to fetch up to a maximum of the specified number of rows. Setting the value lower than the maximum can reduce the response time for returning the initial data. Consider using a smaller value for interactive applications only.

Notes

WSFetchSize and FetchSize can be used to adjust the trade-off between throughput and response time. Smaller fetch sizes can improve the initial response time of the query. Larger fetch sizes can improve overall response times at the cost of additional memory.

Default Value

`2000`

See also

Paging on page 215

Web Service Pool Size

Attribute

WSPoolSize (wsps)

Purpose

Specifies the maximum number of sessions the driver uses. This allows the driver to have multiple web service requests active when multiple JDBC connections are open, thereby improving throughput and performance.

Valid Values

x

where:

x

is an integer that determines the number of sessions the driver uses to distribute calls. This value should not exceed the number of sessions permitted by your account.

Notes

- You can improve performance by increasing the number of sessions specified by this option. By increasing the number of sessions the driver uses, you can improve throughput by distributing calls across multiple sessions when multiple connections are active.
- The maximum number of sessions is determined by the setting of Web Service Pool Size for the connection that initiates the session. For subsequent connections to an active session, the setting is ignored and a warning is returned. To change the maximum number of sessions, close all connections using the driver; then, open a new connection with desired limit specified for this option.

Default Value

`1`

Web Service Retry Count

Attribute

WSRetryCount (wsrc)

Purpose

Specifies the number of times the driver retries a timed-out Select request. The timeout period is specified by the Web Service Timeout (WSTimeout) option.

Valid Values

`0` | *x*

where:

x

is a positive integer

Behavior

If set to `0`, the driver does not retry timed-out requests after the initial unsuccessful attempt.

If set to *x*, the driver retries the timed-out request the specified number of times.

Default Value

`5`

See also

Web Service Timeout on page 197

Web Service Timeout

Attribute

WSTimeout (wst)

Purpose

Specifies the time, in seconds, that the driver waits for a response to a web service request.

Valid Values

`0` | *x*

where:

x

is a positive integer that defines the number of seconds the driver waits for a response to a web service request.

Behavior

If set to 0, the driver waits indefinitely for a response; there is no timeout.

If set to *x*, the driver uses the value as the default timeout, measured in seconds, for any statement created by the connection.

If a Select request times out and Web Service Retry Count (WSRetryCount) is set to retry timed-out requests, the driver retries the request the specified number of times.

Default Value

120

See also

Web Service Retry Count on page 197

7

Model file syntax

The driver employs a Model file to map JSON responses to the relational model. Although the primary purpose of the Model file is to define endpoint and table mapping, it is also capable of configuring a number of driver behaviors, such as paging, custom authentication, and HTTP response code processing. This reference describes the syntax used to configure the features and functionality supported by the Model file.

The Model file is a simple text file that uses the `*file_name*.rest` naming convention. To configure the file, you will need to populate its contents. You can do this using a text editor or by generating a file using the Configuration Manager. For more information on using the Configuration manager to generate a Model File, see "Generating a Model file with the Configuration Manager."

The following is the basic structure of the Model file:

```
1 {
2    "#http":[<http_response_codes>],
3
4    "#<oauth2_param>":"<oauth2_value>",
5
6    "#authentication":[<custom_auth>],
7    "#reauthentication":[<custom_auth>],
8
9    "<table_name1>":"<table_definition1>",
10   "<table_name2>":"<table_definition2>",
11   "<table_name3>":"<table_definition3>"
12
13   "routines":["<function_definition1>",
14               "<function_definition2>",
15               "<function_definition3>"
16   ]
17 }
```

Note: With the exception of table definitions, all REST entries described in the following table are optional.

Table 27: Model file components

Lines	Entry/Entry Type	Description
2	`#http`	Defines how HTTP response status codes are processed by the driver. For details and syntax, see HTTP response code processing on page 201.
4	OAuth 2.0 entries	Configures OAuth 2.0 authentication behavior using a set of entries. This allows you to centrally set and manage OAuth authentication properties for all connections using the file. Note that these entries are mutually exclusive with the `#authentication` entry. For details and syntax, see OAuth 2.0 authentication on page 204.
6	`#authentication`	Defines custom authentication requests that retrieve and exchange access tokens. Custom authentication is used when your service does not support one of the standard authentication methods provided by the driver. Note that this entry is mutually exclusive with the OAuth 2.0 entries. For details and syntax, see Custom authentication requests on page 206.
7	`#reauthentication`	Defines the request used to refresh the access token retrieved through the `#authentication` entry. For details and syntax, see Custom authentication requests on page 206.
9-11	table entries	Defines the tables and columns that are derived from REST endpoints. This section can be used to configure multiple aspects of the driver's behavior, including paging, data type mapping, and filtering. For details and syntax, see Table definition entries on page 209.
13-16	User defined functions and procedures	Defines the user or Progress defined functions and procedures that can be called by the driver. For details and syntax, see User-defined functions and procedures on page 244.

For details, see the following topics:

- HTTP response code processing
- OAuth 2.0 authentication
- Custom authentication requests

- Table definition entries
- User-defined functions and procedures
- URL-encoded values
- Example Model file

HTTP response code processing

The driver allows you to customize how HTTP response status codes are processed by the driver. This provides you with a method to define error responses for codes that are returned by the service, including subsequent driver actions and error messages. By using the `#operation` and `#match` properties, you can further limit the definition to apply only to responses for certain operations or for service responses that contain specific content, such as a specific error message.

If no `#http` entry is created to define how response codes are processed, the driver handles response codes according to the default actions in the following table. Note that if you supply your own `#http` entry, you must provide a definition for each response code that the service returns. Any code not explicitly defined in your entry will be returned as a failure.

Code	Action
200	OK
400	ZERO_ROWS
401	REAUTHENTICATE
404	ZERO_ROWS
429	RETRY_AFTER
503	RETRY_AFTER

An entry to define HTTP response code processing takes the following form for multiple definitions:

```
"#http":[
    {
        "#code":<code_number1>,
        "#action":"<action>",
        "#operation":"<operation>",
        "#match":"<match_string>",
        "#message":"<message>"
    },
    {
        "#code":<code_number2>,
        "#action":"<action>",
        "#operation":"<operation>",
        "#match":"<match_string>",
        "#message":"<message>"
    },
    {
        "#code":<code_number3>,
        "#action":"<action>",
        "#operation":"<operation>",
        "#match":"<match_string>",
        "#message":"<message>"
    }
]
```

Important: The driver reads definitions in the order listed and uses the first one that matches the response being evaluated. Therefore, if you have multiple definitions for a single code response, it's important to specify definitions with `#match` and `#operation` parameters before definitions without conditions. This helps the driver avoid using a definition that is designed to generally apply to a code before evaluating those with specific conditions.

code_number

is the numeric HTTP status code for which you want to define driver behavior. For example, `200`, `401`, `404`, etc.

action

is the action the driver takes after the specified HTTP status code is returned and the values of the `operation` and `match` properties are determined to apply. A list of supported actions is described in the "Action Values" table.

operation

(optional) is the operation types to which you want to limit the response definition to apply. For example, if you only wanted the behavior defined in this entry to apply to instances when performing insert operations, set this value to `Select`. See the "Operation Values" table for a list of supported values and their definition.

match

(optional) is text used to limit the instances to which the response definition applies. When a value is provided for this property, the driver scans the first 512 bytes of the service response for the specified value. If it's detected, the driver determines that the behavior in the definition applies. For example, if you only wanted the behavior to apply when the response body contained the text `"status":"error"`, you would specify the following:

```
"#match":"\"status\":\"error\"",
```

message

(optional) is the message text that you want the driver to return when encountering the specified status code. This is typically the error message you want returned to the user. You can specify this value as plain text or as a reference to a header of the server response. For example, to reference the "message" header in a service response, you would specify the following to display the text contained in the "message" header:

```
"#message":"{message}"
```

Table 28: Action Values

Value	Behavior
OK	The operation was successful without errors. In the case of executing a SELECT, zero or more rows were returned. No further action is taken.
ZERO_ROWS	Similar to OK, the operation was successful without errors; however, it does not try to find any rows in returned content. This can be used to sync the HTTP response behavior of the REST service to the expected SQL behavior. For example, when a URL is designed to fetch multiple objects, but there are no objects of that type to return, some services return a code 200 and an empty array. However, in that same scenario, other services might return a code 404 as an error to signify that no rows were returned. In those instances, you would want the code 404 returned using the ZERO_ROWS action, instead of as an error.
RESET	The driver reinitializes the connection as if it were the first statement
REAUTHENTICATE	The driver tries to authenticate again. This action can be used when an access token expires and a new one needs to be fetched.
RETRY_AFTER	The driver retries the query up to the number of times specified by the wsretrycount option[1]. Note that if the response includes a Retry-After header, the driver will honor it.
RETRY_ONCE	The driver retries the operation once.
RETRY_GOOGLE	The driver retries the operation up to the number of times specified by the wsretrycount option[1] or 5 times, whichever is lesser, using the Google exponential backoff rules.
RETRY_AWS	The driver retries the operation up to number of times specified by the wsretrycount option[1], using the Amazon Web Services exponential backoff rules as implemented in the `getWaitTime()` method.
RETRY_FIXED	The driver attempts to retry the operation up to the number of times specified by the wsretrycount option[1]. The number of milliseconds between attempts is set by the `PhoenixMetaData.setFixedRetryDelay(int)` method. The default setting is `1` second.
FAIL	The operation should throw an exception. For example, a code 400 might mean that the service failed to comprehend the query.

[1] wsretrycount is an unexposed option. You can specify a value for this option using the Extended Options field, `odbc.ini`, `odbcinst.ini`, or connection string. The default is 5 attempts (`wsretrycount=5`).

Table 29: Operation Values

Value	Description
`SELECT`	All API calls involved in sampling endpoints or querying rows
`LOGIN`	API calls related to logging in, such as those for authenticating with OAuth2. Note that credentials are *not* automatically added to these requests.
`API`	API calls not related to data, such as those to retrieve schema or user settings.
`GOODBYE`	API calls related to logging out without overriding any result status actions like OK.

OAuth 2.0 authentication

The Model file supports a set of entries that can be used for OAuth 2.0 authentication. As opposed to specifying these values in a connection string or data source, using a Model file allows you to centrally configure and manage certain OAuth 2.0 settings for all connections using that file.

Note: The OAuth 2.0 authentication entries described in this section are mutually exclusive from `#authentication` entry, which is used for custom authentication flows.

The following demonstrates the syntax used for specifying OAuth 2.0 settings in the Model file. Note that different authentication flows, or grant types, require a different set of credentials and authentication locations to successfully authenticate. Therefore, not all of these entries will be used for every flow. If you are unsure of your requirements, contact your system administrator.

Note: Entries that correspond to connection options that specify confidential information, such as Client ID (ClientID) and Client Secret (ClientSecret), are not supported in the Model file. Values for these options should be passed in a connection string or by the application.

```
"#authenticationMethod":"OAuth2"
"#authUri":"<auth_uri>"
"#enableLoginPrompt":true | false
"#logoffUri":"<log_off_uri>"
"#redirectUri":"<token_uri>"
"#scope":"<scope>"
"#tokenUri":"<token_uri>"
```

Table 30: Supported Auth2.0 entries

Entry	Description
`#authenticationMethod`	Determines which authentication method the driver uses during the course of a session. Set this value to `OAuth2`.
`#authUri`	Specifies the endpoint for obtaining an authorization code from a third-party authorization service
`#enableLoginPrompt`	Specifies whether the driver fetches access and refresh tokens at connection when logon credentials are provided via the login prompt for your service. Set this option to `true` if you are using dynamic authorization code grant flow.
`#logoffUri`	Specifies the endpoint the driver calls to notify the service to log the client out of the session, including performing any clean-up tasks or expiring the token.
`#redirectUri`	Specifies the endpoint to which the client is returned after authenticating with a third-party service.
`#scope`	Specifies a space-separated list of OAuth scopes that limit the permissions granted by an access token.
`#tokenUri`	Specifies the endpoint used to exchange authentication credentials for access tokens. For example, `https://example.com/oauth2/authorize/`.

Examples

The following examples demonstrate potential entries for common authentication flows.

Authorization code grant:

```
"#authenticationMethod":"OAuth2"
"#redirectUri":"http://localhost"
"#tokenUri":"https://example.com/oauth2/token"
```

Client credentials, Password, and Refresh token grants:

```
"#authenticationMethod":"OAuth2"
"#tokenUri":"https://example.com/oauth2/token"
```

Custom authentication requests

Note: In addition to directly modifying the Model file, you can also specify the entries discussed in this section using the Autonomous REST Composer. See Configuring custom authentication with the Configuration Manager on page 78 for more information.

If your service does not support one of the standard authentication methods provided by the driver, you can define custom authentication requests to retrieve and exchange access tokens using the Model file. Multiple authentication requests can be defined in a single entry, allowing you to implement authentication flows that consist of multiple steps.

A custom authentication request is defined in the Model file using two entries:

- `#authentication`: defines the initial request in an authentication flow.
- `#reauthentication`: defines the request used to refresh the access token retrieved through the `#authentication` entry.

Important: In authentication request entries, special characters, such as ampersands (`&`), must be escaped using a back slash (`\`).

The `#authentication` and `#reauthentication` entries are comprised of the following components:

- **Header**: Defines the header to be included in the HTTP request used to retrieve an access token. The header applies to the next HTTP request defined in the entry. A header must be defined for each HTTP request that retrieves a token.
- **Payload**: Contains the request body, in JSON format, that must be passed to the service to generate the access token. The payload applies to the next HTTP request defined in the entry. A payload should be defined only if a request body is required by the service for that HTTP request.
- **HTTP request**: Defines the HTTP request to the endpoint that is used to exchange authentication credentials for access tokens. An HTTP request must be defined each time a token is retrieved.
- **Data request credentials**: Defines the header or parameter used in requests for data. The data request credentials take the following form, where *service_reponse* is the service response containing the access token:

  For headers:

  ```
  "HEADER <header_name>=<header_value> {/<service_response>}"
  ```

  For parameters:

  ```
  "PARAM <parameter_name>=<parameter_value> {/<service_response>}"
  ```

Modifying unique parameters and credentials

To allow you to modify parameter values and payloads on a per connection basis, the Model file supports using variables to reference connection option values specified in the connection string or data source definition. This provides you with a secure method to specify unique values for each connection without having to edit the Model file. For most options, you can create a variable by enclosing the option attribute name in `{ }` brackets. For example, to reference the value of the Password option in the connection string, specify `{password}`.

The exception to this behavior is the Custom Authentication Parameters (CustomAuthParams) option. The value of the Custom Authentication Parameters options is a semicolon-separated list of parameter values. To indicate the correct value in the list, in addition to the attribute name, you must also specify the ordinal location of the parameter you want to reference in `[ ]` brackets. For example, to reference `www.example.com` in the following Custom Authentication Parameters value, you would use a variable of `{CustomAuthParams[3]}`.

```
CustomAuthParams=123XYZ456abc789;My Company Inc;www.example.com
```

Note:

- The property name variables enclosed in brackets are case insensitive. For example, both `{password}` and `{Password}` reference the Password connection option.
- If you specify a property name that does not resolve, the driver returns an error when attempting to issue a request. For example, this could occur if the property specified is not supported or if there is a typographical error in the specified variable.
- When using Custom Authentication Parameters variables, if the specified ordinal position does not correlate to a value in the Custom Authentication Parameters connection option, the driver returns an error when attempting to issue a request. For example, if specifying `{CustomAuthParams[3]}`, but only two values are specified by the connection property, such as `CustomAuthParams=123XYZ456abc789;My Company Inc`.

Base64 encoded values

If basic authentication is required for your custom authentication, you can configure the driver to calculate Base64-encoded user name and password values at runtime. Use the following syntax in the payload to use Base64 encoding:

```
"Authorization=Basic BASE64({user}:{password})",
```

When issuing a request, the driver encodes the values specified for the user and password connection options and uses them to authenticate to the service. If Base64 encoding is not used, the driver passes the user and password values in plain text.

Examples

The following are some common examples using custom authentication:

- Authorization header authentication
- Simple token request
- Two-step token request

Example: Authorization header authentication

The following example demonstrates a simple form of authentication where three headers are sent to authenticate to the service. Unlike other examples in this section, there is no request for access tokens.

In this example, the authorization header must pass a value that contains `Basic` followed by a Base64-encoded value set of the user name and password. Because the example is using variables for the user and password, the values are supplied by the setting of the User and Password connection options. The following two headers pass hard-coded values for the client ID and client secret. These values are provided using arguments in the customAuthParams connection option value.

```
"#authentication": [
//Authorization header
    "HEADER Authorization=Basic (BASE64)({user}/{password})",
//Client ID value specified as the first argument of the customAuthParams connection
//option
    "HEADER X-Client-Id={customAuthParams[1]}",
//Client secret value specified as the second argument of the customAuthParams connection

//option
    "HEADER X-Client-Secret={customAuthParams[2]}"
]
```

Example: Simple token request

The following is an example of a simple token request, where `access-token` is the server response that contains the payload with the access token. Most custom authentication requests will take this form.

```
"#authentication" : [
//Header
    "api-key={CustomAuthParams[1]}",
//Payload
    {
        "credentials": {
            "username": "{user}",
            "password": "{password}",
            "company": "{customAuthParams[2]}"
        }
    },
//HTTP request
    "POST http://{serverName}/bearertoken",
//Data request credentials. "{/access-token}" refers to the service
//response from the preceding HTTP request.
    "HEADER Authentication=Bearer {/access-token}"
]
```

Example: Two-step token request

The following example demonstrates a two-step authentication, where the service response from the initial request, `UserToken`, is passed in the request header of the second stage of authentication. The principles demonstrated in this example apply to authentication flows requiring two or more requests.

```
"#authentication" : [
//Header request for first request
    "accept=application/json",
    "content-type=application/json",
    "kmauthtoken=\\{sitename:\"{customAuthParams[1]}\",localeId:\"en_US\"\\}\\}",
//Payload for first request
    {
        "login": "{user}",
        "password": "{password}",
        "siteName": "{customAuthParams[1]}"    },
//HTTP request for first token
    "POST https://{serverName}/getusertoken",
//Header for second request. "{/authenticationToken}" refers to the value of
//the service response from the preceding HTTP request.
    "accept=application/json",
    "content-type=application/json",
    "kmauthtoken=\\{sitename:\"{CustomAuthParams[1]}\",localeId:\"en_US\",
     UserToken:\"{/authenticationToken}\"\\}",
```

```
//Payload for second request
    {
        "userName": "{user}",
        "password": "{password}",
        "siteName": "{customAuthParams[1]}",
        "userExternalType": "ACCOUNT"
    },
//HTTP request for second token
    "POST https://{serverName}/getaccesstoken",
//Data request credentials. "{/customAuthToken}" refers to the value in
//the service response from the preceding HTTP request.
    "HEADER Authentication=Bearer
        \\{sitename:\"{customAuthParams[1]}\",localeId:\"en_US\",
           userToken:\"{/authenticationToken}\",integrationUserToken:\
           "{/accessToken}\"\\}"]
```

Table definition entries

Table definition entries define the mapping of JSON endpoints to tables. You can specify a single entry or, in a comma separated list, multiple entries. These entries can be as simple as a colon-separated table name and endpoint pair (`"<table_name>":"<endpoint>"`,). Or, for a greater level of configuration, entries can take the form of JSON objects.

The following demonstrates the syntax of a set of three simple table definition entries. For endpoint details and syntax, see "Query paths."

```
"<table_name1>":"<endpoint1>",
"<table_name2>":"<endpoint2>",
"<table_name3>":"<endpoint3>"
```

The following demonstrates the syntax used to configure a single table entry in an array.

Note: The following example demonstrates the syntax for all the features and functionality supported by the driver, but it is not typical for defining a table. In most scenarios, only a subset of these parameters would be used to define a table.

```
1 {
2   "<schema_name>.<table_name>": {
3      "#path":["<endpoint>"],
4      "#insert": "<method> <endpoint>",
5      "#update": "<method> <endpoint>",
6      "#delete": "<method> <endpoint>",
7      "#<paging_parameter>":"<paging_value>",
8      "#<parsing_parameter>":"<parsing_value>",
9   // The following POST entry defines two fields.You can define one or more
10  // fields in an entry.
11     "#post":{"<field1>":"<value1>","<field2>":"<value2>"}
12  //The following HTTP header entry defines two headers. You can define one or more
13  //headers in an entry.
14     "#headers":{"<header1>":"<value1>","<header2>":"<value2>"}
15     "<column1>":"<data_type>",
16     "<column2>":"<data_type>,#key",
17     "<column3>":"<data_type>,#readonly,
18     "<column3>":"<data_type>,#notnull,
19  //The following array defines two nested columns. You can define one or more
20  //nested columns in an array.
21    "<column3>[]":{"<column_a>:<data_type>","<column_b>":"<data_type>"}
22  //The following key-map defines two columns. You can define one or more
23  //columns in a key map.
24    "<column4>{<data_type>}":{"<column_c>:<data_type>","<column_d>":"<data_type>"}
25  //The following column defines two nested objects. You can define one or more
26  //nested columns in table definition.
27     "<column5>":{"<nested_column1>":"<data_type>","<nested_column2>":"<data_type>"}
28     "<column6>":"<data_type>","<java_date_format>"
29     "<column7>":{"#type":"<data_type>","#extract:<reg_expression>"}
30     "<column8>":{"#type":"<data_type>","#header":true,"#eq":"<header_name>"}
31     "<column9>":{"#type":"<data_type>","#<operator>":"<uri_property>"}
32     }
33 }
```

Table 31: Components of a table definition entry

Lines	Entry/Entry Type	Description
2	Table name	(Required) Specifies the name of the table. Optionally, you can specify the name of your schema. For details and syntax, see Schema name on page 212.
3	`#path`	(Required) Specifies the query path to an endpoint(s) that the driver connects to and samples. This can be a full endpoint, the path portion of an endpoint, or an array of endpoints. For details and syntax, see Query paths on page 230.
4-6	Write operations	Configures write operations for the specified endpoint values. Without these entries, the resulting relational table will be read only. For details, see Write operations on page 213.

Lines	Entry/Entry Type	Description
7	Paging parameters	Configures paging behavior for the table using a set of parameters. These parameters differ based on the paging mechanisms you want to employ. For details and syntax, see Paging on page 215.
8	Parsing parameters	Configures the parsing behavior of the driver using a set of parameters. This allows the driver to accurately parse services that do not use pure REST syntax, such as legacy or proprietary services. For details and syntax, see REST model parsing on page 223.
11	`#post`	Defines the sample values used when issuing a POST request. For details and syntax, see POST requests on page 224.
14	`#headers`	Specifies the HTTP headers to filter data returned by a request. For details and syntax, see Requests with custom HTTP headers on page 228.
15-29	Column definitions	Defines the name of the column and additional mapping. Column names can be literal or regular expressions. You can also configure data type mapping in these fields. For details and syntax, see Column names on page 234 and Data type mapping on page 235.
16	Primary key	Designates the primary key by specifying the `#key` element in a column definition. For details and syntax, see Primary key on page 237.
17-18	Column flags (read only and nullable)	Designates a column as read only by specifying the `#readOnly` element in a column definition. In addition, you can use the `#notNull` element to set the nullable flag for metadata purposes. For details and syntax, see Read-only columns on page 237 and Nullable columns on page 238.
21	Column as an array	Defines a column as an array by specifying brackets (`[]`) at the end of its column name. For details and syntax, see Columns as an array on page 238.
24	Column as key-value map	Defines a column as an key-value map by specifying brackets (`{}`) at the end of its column name. For details and syntax, see Columns as a key-value map on page 239.
27	Column with nested objects	Defines a column with nested objects in the entry body. For details and syntax, see Columns with nested objects on page 239.

Lines	Entry/Entry Type	Description
28	Time stamp formats	Defines the time stamp format for a column in the definition. For details and syntax, see Date, time, and timestamp formats on page 240.
29	#extract	Specifies a regular expression that allows you to extract a subfield, or portion, of a string value. For details and syntax, see Subfields on page 241.
30	#header	Specifies whether the column can be sent as an HTTP header instead of part of a query string for GET requests. For details and syntax, see Columns as HTTP headers on page 241.
31	Filtering and URI parameters	Specifies filtering operations to be sent in requests for the column. For details and syntax, see Filtering and URI parameters on page 242.

Schema name

You can change the name of your schema by modifying the Model file. In the table object, specify the schema name you want to use before the table name and separated by a period. If no schema name is specified, tables are mapped to the AUTOREST schema by default.

Table entries that specify user-provided schema names take the following form:

```
"<schema_name1>.<table_name1>":"<host_name1>/<endpoint_path1>",
"<schema_name2>.<table_name2>":"<host_name2>/<endpoint_path2>"[, ...]
```

schema_name

is the name of the schema to which the driver maps the table. For example, MYSCHEMA.

table_name

is the name of the relational table to which the driver maps the endpoint. For example, countries.

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, http://example.com. You can omit this value by specifying the host name using the Host Name (HostName) option.

endpoint_path

is the path component of the URL endpoint. For example, country. This can be an unparameterized or parameterized path, a path that uses query parameters, or an array of paths. See "Query paths" for examples and more information.

In the following example, the `employees` table maps to the `MYSCHEMA` schema, and, because no schema was specified for the table, the `countries` table maps to `AUTOREST` schema by default:

```
"myschema.employees":"https://example.com/employees",
"countries":""http://example.com/countries/"
```

See also

Query paths on page 230

Write operations

To execute Insert, Update, and Delete statements against your data, you must first configure the corresponding write operation directives in the table definition. The values specified for the directives contain the REST verb to use for the operation and the endpoint for which you want the corresponding write operation enabled. For example, if you want to enable Insert statements for your endpoint, you must specify your endpoint using the `#insert` directive. See "Insert", "Update", and "Delete" for more information on SQL statement syntax.

Note: When enabling write operations for an endpoint, you can designate columns mapped from that endpoint as read-only using the `#readOnly` directive. Designating columns as read-only prevents data stored within from being inadvertently changed. See "Read-only columns" for examples and more information.

A table entry with insert, update, and delete operations enabled would take the following form:

```
"<table_name>": {
      "#path": ["<host_name_1/<endpoint_path_1>","<host_name_2/<endpoint_path_2>",...],

      "#insert": "<http_method> <host_name>/<operation_path>",
      "#update": "<http_method> <host_name>/<operation_path>",
      "#delete": "<http_method> <host_name>/<operation_path>",
      <column_definitions>
 },
```

Note that if your endpoint definitions require that you pass the values to be inserted or updated in an array in the POST body, then add square brackets (`[ ]`) to the endpoint definition in the write operation directive:

```
"<table_name>": {
       "#path": ["<host_name_1/<endpoint_path_1>","<host_name_2/<endpoint_path_2">,...],

        "#insert": "<http_method> <host_name>/<operation_path>[]",
        "#update": "<http_method> <host_name>/<operation_path>[]",
        <column_definitions>
 },
```

table_name

is the name of the relational table to which the driver maps the endpoint. For example, `countries`.

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the ServerName property.

endpoint_path

is the path component of the URL endpoint. For example, `country`. This can be an unparameterized or parameterized path, a path that uses query parameters, or an array of paths. See "Query paths" for examples and more information.

http_method

(optional) is the HTTP method used to perform the request for the specified write operation. By default, this method is `POST` for inserts, `PUT` for updates, and `DELETE` for deletes. The default values will work for many REST services; however, you may need to specify a different method according to the expectations of your API.

Note that for update operations, certain APIs require `PATCH`, `PUT`, or `POST` methods to send only updated fields in the `POST` body. This is the expected behavior for the `PATCH` method; therefore, if your API uses `PATCH`, you only need to specify `PATCH` as your HTTP method to send only updated fields. For example:

```
"#update": "PATCH http://example.com/countries/{id}",
```

However, the typical behavior for the `PUT` or `POST` methods is to overwrite data, effectively deleting your existing data and replacing it with an updated version. To configure the `PUT` or `POST` methods to send only updated fields, you must also set the `#sendOnlyUpdated` parameter to `true`. For example, the following will send only the updated fields using the `PUT` method:

```
#update": "http://example.com/countries/{id}",
"#sendOnlyUpdated":"true"
```

operation_path

is the path component of the URL endpoint against which a write operation can be performed. For example, `country`. This can be an unparameterized or parameterized path, a path that uses query parameters, or an array of paths. See "Query paths" for examples and more information.

For example, the following entry enables Insert and Delete statements to be executed against data in the specified endpoint and the `Countries` table.

```
"Countries": {
        "#path": ["http://example.com/countries/",
                   "http://example.com/countries/{id}"],
        "#insert": "http://example.com/countries/",
        "#delete": "http://example.com/countries/{id}",
        "id":"VarChar(32),#key,"
        "name":"VarChar(46)",
        "population":"Integer"
    },
```

The following example demonstrates an entry that enables the driver to insert the contents of an array element in a POST body.

```
"Countries": {
        "#path": ["http://example.com/countries/",
        "http://example.com/countries/{id}"],
        "#insert": "http://example.com/countries/{id}[]",
        "id":"VarChar(32),#key",
        "name":"VarChar(46)",
        "population":"Integer"
    },
```

Passing the primary key in the body of a request

In addition to passing the primary key in the URL endpoint (e.g., `http://example.com/countries/{id}`), some APIs require that the primary key is passed in the body of a request when performing a write operation. You can configure the driver to pass the primary key in the body of an insert or update operation by adding the corresponding operation directive (`#insert` | `#update`) after the `#key` directive. In the following example, the primary key is passed in the body of insert and update operations because the `#insert` and `#update` directives are specified in the column definition of the primary key.

```
{
"Countries":{
          "#path": ["http://example.com/countries/",
                    "http://example.com/countries/{id}"]
          "#insert":"http://example.com/countries/{id}",
          "#update":"http://example.com/countries/{id}",
          "id":"VarChar(32),#key,#insert,#update",
          "name":"VarChar(46),#readOnly",
          "population":"Integer"
          }
},
```

See also

Insert on page 271
Update on page 282
Delete on page 270
Read-only columns on page 237

Paging

The connector supports the following paging mechanisms:

- Row offset paging
- Page number paging
- Next page token

To configure paging, specify values for the following properties that correspond to the mechanism you want to employ. These properties can be specified at either the top level of the Model file, as an entry, or as a property in the body of a table definition. Properties set at the top level define the default behavior for all the tables defined in the file, while properties specified in a table definition override paging behavior for that table. If paging properties are not specified, the driver attempts to retrieve the first page for data sources that require paging.

In addition, for data sources that support more complicated parameters, you can specify parameters using a template. See "Using templates for paging parameters" for details.

Note that you can also configure paging to use HTTP response headers. See Paging that uses HTTP headers for examples of Model file syntax.

The following demonstrates the syntax used to configuring row offset paging in the body of a table definition:

```
"<table_name>": {
          "#path": "<host_name>/<endpoint_path>",
          "#firstRowNumber":1,
          "#maximumPageSize":1000,
          "#pageSizeParameter":"maxResults",
          "#rowOffsetParameter":"startAt"
 },
```

General paging parameters

The following table describes optional parameters that can be used by all paging mechanism types:

Table 32: General Paging Properties

Property	Description

`#fieldListParameter`	Specifies the name of the URI parameter that contains the comma separated list of fields to be issued in a request. For example, if you were to issue the following query with `#fieldListParameter=fields`:
	`SELECT * FROM ORDERS`
	The following request would be issued for an entry containing `id` and `success` fields:
	`https://www.example.com/ORDERS?fields=id%2Csuccess`

`#hasMoreElement`	Specifies the element in the response that denotes there is another page. The service indicates there are no additional pages by omitting the element from the response or returning a value of false. For elements not stored at the top level, this value should include a slash-separated path.
`#maximumPageSize`	Specifies the maximum page size in rows.
`#pageSizeElement`	Specifies the name of the element containing the page size in rows that must be passed in the URI to get the next page. For elements not stored at the top level, this value should include a slash-separated path.
`#totalPagesElement`	Specifies the name of the element in the response that contains the total number of pages contained in the result set. For elements not stored at the top level, this value should include a slash-separated path.
`#totalRowsElement`	Specifies the name of the element in the response that contains the total number of pages contained in the result set. For elements not stored at the top level, this value should include a slash-separated path.

Row offset paging

The following table describes the parameters that are specific to configuring row offset paging:

Table 33: Row Offset Paging Properties

Property	Description
`#firstRowNumber`	Specifies the number of the first row. The default is `0`; however, some systems begin numbering rows at 1.
`#pageSizeParameter`	Specifies the name of the URI parameter that contains the page size.
`#rowOffsetParameter`	Specifies the name of the URI parameter that contains the starting row number for this set of rows.

Page number paging

The following table describes the parameters used to configure page number paging:

Table 34: Page Number Paging Properties

Property	Description
`#firstPageNumber`	Specifies the number of the first page. The default is `0`; however, some systems begin numbering pages at 1.
`#pageSizeParameter`	Specifies the name of the URI parameter that contains the page size.
`#pageNumberParameter`	When requesting a page of rows, this is the name of the URI parameter to contain the page number.

Next page token paging

The following table describes the parameters used to configure next page token paging. Note that next page token paging also supports paging for the following:

- APIs that return paging parameters as a URL, header, or query parameter in a response body.
- APIs that return paging parameters as a URL, header, or query parameter in a response header.
- APIs that use a query parameter to determine what data value to start after when returning the next page of results. See Example: Paging that uses the starting after scenario on page 222 for syntax examples.

Table 35: Next Page Token Paging Properties

Property	Description
#nextPageElement	Specifies the name of the element name of the element in the current response that contains the token that must be passed in the URI to get the next page. For elements not stored at the top level, this value should include a path to the element. If your API returns the URL, query parameter value, or HTTP header value used paging in the body of the response, set the #nextPageElement to specify the element in the response containing the applicable value. If your API utilizes starting after paging, set the #nextPageElement directive to specify the field in the response that contains the data value to be sent in the query parameter to request the next page of results. This value should take the following form <object_name>/<field_name>.
#nextPageParameter	Specifies the name of the URI parameter that holds the token used to fetch the next page. This is the token found on the current page at the location specified by the #nextPageElement. For a starting after scenario, you would specify the name of the query parameter that the driver needs to use to tell the API which data value to start after when returning the next page of results.
#pageSizeParameter	Specifies the name of the URI parameter that contains the page size.
#hasMoreElement	Specifies the element in the response that denotes there is another page. The service indicates there are no additional pages by omitting the element from the response or returning a value of false. For elements not stored at the top level, this value should include a slash-separated path.
#nextPageRequestHeader	Specifies the name of the HTTP header to be specified in a request to retrieve the next page of results.
#nextPageResponseHeader	Specifies the name of the header in the response that contains the value used for retrieving the next page of a result. This value can be either a URL, a value argument parameter in a query parameter, or a value argument of an HTTP header that is issued in request to return the next page.

Example: Paging that uses HTTP link headers

The following examples demonstrate the Model file syntax for services that configure paging by passing parameters in headers.

GitHub

When paging is configured in a request to GitHub, the service includes link headers in the response header. Link headers are comprised of a list of URLs with query parameters that can be used to fetch pages in a result. Through the query parameters, link headers configure pagination behavior such as page size and ordinal information. For example:

```
link: <https://api.github.com/repositories/1234567/issues?page=2>; rel="prev",
      <https://api.github.com/repositories/1234567/issues?page=4>; rel="next",
      <https://api.github.com/repositories/1234567/issues?page=200>; rel="last",
      <https://api.github.com/repositories/1234567/issues?page=1>; rel="first"
```

To enable paging and set parameters in a request to GitHub, you must to specify certain properties in the Model file. First, set the `#maximumPageSize` property to specify the maximum rows returned in a page. Then, configure the `#nextPageParameter` to specify which URI parameter that holds the token to fetch the next page.

The following example demonstrates configuring next page token paging for the `AllUsers` table from a GitHub service.

```
"AllUsers":{
 "#maximumPageSize":"30",
 "#nextPageParameter":"since",
 "#path":[
  "IDENTITY users/{login}",
  "users"
 ],
 "#headers": {
  "Accept": "application/vnd.github.v3+json"
 },
 "id":"Integer,#key",
//Additional table definition syntax can be populated after paging parameters.
}
```

OKTA

For OKTA services, responses to requests that configure paging include a series of link headers that are used to fetch pages in the result. Each link header returned represents a page in the result and includes paging configuration information, such as the page size, ordinal information, and the cursor ID number. For example:

```
link: <https://{myDomain}/api/v1/logs?limit=25>; rel="self"
link: <https://{myDomain}/api/v1/logs?limit=25&after=1234567898765_1>; rel="next"
```

To configure paging for OKTA services in a request, you must specify the page size using the `#maximumPageSize` property and, via the `#nextPageParameter`, the query parameter specifying the token ID number.

The following example demonstrates configuring Next Page Token Paging with the `USERS` table from an OKTA service.

```
"USERS": {
    "#path": [
      "/users"
    ],
 "#maximumPageSize": 20,
 "#nextPageParameter": "after",
 "id": "VarChar(30),#key",
//Additional table definition syntax can be populated after paging parameters.
}
```

Example: Next page token paging that uses a URL returned in a response body

Some services employ a paging mechanism that returns the URL for the next page of results in response bodies. For example, an element in a response body that specifies the URL takes the following form:

```
"next": "https://example.com/myEndpoint?nextpage=2"
```

In this scenario, to configure paging, specify the name of the element in the response body using the `nextPageElement` parameter. For example, the following entry configures the driver to use the value of the `next` element in the response body to retrieve the next page in the results:

```
"#pageSizeParameter": "limit",
"#maximumPageSize": 100,
"#nextPageElement": "next"
```

Example: Next page token paging that uses an HTTP header returned in a response body

This section describes how to configure the driver for paging mechanisms that use an HTTP header returned in a response body to return the next page of results. In the following example, the response body contains an element, `next`, that specifies the HTTP-header value argument, `page_2`, to be passed in the next request.

```
"next": "page_2"
```

If the HTTP header used to return the next page of a result is `next_page`, the driver would send the following HTTP header argument in a request to retrieve the next page of results.

```
next_page=page_2
```

When configuring your paging entry for this mechanism, you need to specify the `nextPageElement` and `nextPageRequestHeader` parameters. The `nextPageElement` specifies the element in the response body that contains the value of the HTTP header, while `nextPageRequestHeader` specifies the HTTP header to be used to retrieve the next page of results. The following is example of configuring paging using a query parameter passed in a response.

```
"#pageSizeParameter": "limit",
"#maximumPageSize": 100,
"#nextPageElement": "next",
"#nextPageRequestHeader": "next_page"
```

Example: Next page token paging that uses a query parameter returned in a response body

The driver supports services that employ a paging mechanism that use query parameters passed in response bodies to return the next page of results. In the following example, the response body contains an element, `next`, that specifies the query parameter value, `page_2`.

```
"next": "page_2"
```

If the query parameter argument is `next_page`, the driver would send the following query parameter in a request to retrieve the next page of results.

```
next_page=page_2
```

To configure paging in this scenario, you need to specify the `nextPageElement` and `nextPageParameter` parameters in your paging entry. The `nextPageElement` configures the driver to use the value of the specified element in the response body as the value of the query parameter, while `nextPageParameter` specifies the query parameter argument to be used to retrieve the next page of results. The following is example of configuring paging using a query parameter passed in a response.

```
"#pageSizeParameter": "limit",
"#maximumPageSize": 100,
"#nextPageElement": "next",
"#nextPageParameter": "next_page"
```

Example: Next page token paging that uses a URL returned in a response header

The driver supports services that use a paging mechanism that returns the URL for the next page of results in a response header. For example, a response header that specifies the URL takes the following form:

```
Next=https://example.com/myEndpoint?nextpage=2
```

To configure paging, specify the name of the header that contains the URL using the `nextPageResponseHeader` parameter. For example, the following entry configures the driver to use the response header named `Next` to retrieve the next page in the results:

```
"#pageSizeParameter": "limit",
"#maximumPageSize": 100,
"#nextPageResponseHeader": "Next"
```

Example: Next page token paging that uses an HTTP header returned in a response header

In this section, we are going to demonstrate how to configure the driver for paging mechanisms that use an HTTP header returned in a response header to return the next page of a result set. The following is an example of a header that is returned in a response. The header, `next`, specifies the value argument of an HTTP header that is issued in a subsequent request to return the next page in the result set.

```
next=page_2
```

If the HTTP header used to return additional pages is `next_page`, the driver would issue a request with the following HTTP header to retrieve the next page.

```
next_page=page_2
```

To configure paging for this mechanism, you need to specify values for the `nextPageResponseHeader` and `nextPageRequestHeader` parameters. The `nextPageResponseHeader` parameter specifies the name of the header in the response that contains the value of the HTTP header to be used in the subsequent request. In this case, the value is `next`. Conversely, the `nextPageRequestHeader` parameter specifies the name of the HTTP header that issued in the request to return the next page. In this example, the value should be `next_page`. The following is example of configuring paging using an HTTP header passed in a response header.

```
"#pageSizeParameter": "limit",
"#maximumPageSize": 100,
"#nextPageResponseHeader": "next",
"#nextPageRequestHeader": "next_page"
```

Example: Next page token paging that uses a query parameter returned in a response header

This section describes the Model file syntax for paging mechanisms that use query parameters passed in a response header to return the next page of results. In the following example, the response header `Next` specifies the query parameter value `page_2`.

```
Next=page_2
```

If the query parameter argument is `next_page`, the driver would send the following query parameter in a request to retrieve the next page of results.

```
next_page=page_2
```

In this scenario, to configure paging, you need to specify the `nextPageResponseHeader` and `nextPageParameter` parameters in your paging entry. The `nextPageResponseHeader` configures the driver to use the value of the specified header as the value of the query parameter, while `nextPageParameter` specifies the query parameter argument to be used to retrieve the next page of results. The following is example of configuring paging using a query parameter passed in a response.

```
"#pageSizeParameter": "limit",
"#maximumPageSize": 100,
"#nextPageResponseHeader": "Next",
"#nextPageParameter": "next_page"
```

Example: Paging that uses the starting after scenario

For this example, we are going to configure paging for the Stripe service, which requires a value to be set for the `starting_after` query parameter when making requests for the next page of data in the result set. The following is a simple JSON response from a Stripe service:

```
{
  "data": [
    { "id": "50031003", "name": "ABC Company" },
    { "id": "20143243", "name": "General Imports INC." }
  ],
  "has_more" : true
}
```

To configure paging for Stripe, specify the `starting_after` parameter using `#nextPageParameter` property. In addition, you must designate the field that contains the object identifier using the `#nextPageElement` property. Since the identifier field in our response is `id` and is embedded in the `data` object, we would specify the path `/data/id`. The following is an example configuration for the Stripe service:

```
"#pageSizeParameter": "limit",
"#maximumPageSize": 100,
"#nextPageParameter": "starting_after",
"#nextPageElement": "/data/id",
"#hasMoreElement": "/has_more",
```

Using templates for paging parameters

For REST services that use more complicated paging parameters, such as a single URI parameter that contains both an offset and limit parameter, the driver supports using templates to configure paging parameters. In these scenarios, you can specify the pair for the values for the following parameters:

- `NextPageParameter`
- `PageNumberParameter`
- `PageSizeParameter`
- `RowOffsetParameter`

The following syntax specifies templates for paging parameter values:

```
"<paging_parameter>":"<uri_option_name>=<option_element1>:{<token1>}[,<option_element2:<token2>[,...]]"
```

For example, the following demonstrates using variables to configure RowOffsetParameter paging:

```
"<table_name>": {
          "#path": "<host_name>/<endpoint_path>",
          "#maximumPageSize":100,
          "#rowOffsetParameter":"locator=start:{OFFSET},count:{LIMIT}"
 },
```

You can specify one or more of the following templates in the *`option_name=template`* pair:

Table 36: Paging parameter variables

Token	Description
`{LIMIT}`	References the page size.
`{OFFSET}`	References the starting row number.
`{PAGE}`	References the page number.
`{NEXT}`	References the next-page token.

REST model parsing

In addition to supporting the standard REST architecture, the driver supports RESTful or REST-like services. To support idiosyncrasies in certain REST services, the driver includes a set of parsing parameters to adjust how data is being parsed. For example:

```
"#<parsing_parameter>":"<parsing_value>"
```

The following table describes the parsing parameters that are currently supported.

Table 37: Parsing properties

Property	Description

#chunked	Set to `true` if your native JSON objects are not separated by commas, such as with those using the JSON Lines format. For example:
	`{"Name":Sam,"Pet":"dog","vehicle":"car"}`
	`{"Name":Denise,"Pet":"sugar bear","vehicle":"bike"}`
	The default is `false`.

POST requests

To use POST requests, you must define the request in the Model file in the JSON format. The definition entry is comprised of a path and body. The path contains the URL endpoint and the body used in requests, while the body defines documents and provides sample values. The driver then uses these sample values to define which data type to be used when executing a POST request.

Note: The driver also supports issuing POST requests with an empty body. For details, see "POST requests with an empty body."

An entry for a POST request with a parameterized or unparameterized path takes the following form for an entry defining two fields:

```
"<table_name>": {
          "#path": "<host_name>/<endpoint_path>",
          "#post": {
               "<field1>":"<value1>",
               "<field2>":"<value2>"
          }
     },
```

An entry for a POST request with parameters takes the following form:

```
"<table_name>": {
          "#path": "<host_name>/<endpoint_path>",
          "#post": {
               "<field1>":"<value1>",
               "<field2>":"<value2>"
          }
           "<column_name>":{
                  "#type":"<data_type>",
                  "<operator>"":"<uri_parameter>"
          }
     },
```

You can also map custom parameter values to a column using the `#postParameter` property. This allows for filtering in scenarios where complex parameter syntax is employed, such as using complicated JSON data or empty arrays. An entry for a POST request that filters using a custom parameter takes the following form:

```
"<table_name>": {
          "#path": "<host_name>/<endpoint_path>",
          "#post": {
               "<field1>":"<value1>",
               "<field2>":"<value2>"
          }
```

```
            "<column_name>":{
                    "#type":"<data_type>",
                    "#postParameter":"<merge_behavior>",
                    "#default":"<default_parameter>"
            }
      },
```

table_name

is the name of the relational table to which the driver maps the endpoint. For example, `countries2`.

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the ServerName property.

endpoint_path

is the path component of the URL endpoint. For example, `country`. This can be an unparameterized or parameterized path, a path that uses query parameters, or an array of paths. See "Query paths" for examples and more information.

field

is the field name of the *field=value* pair. For example, `START_DATE`.

value

is the sample value the driver uses to determine the data type to use when executing a POST to that document. For example, `2018-08-31`.

column_name

specifies the name of the column against which you are using query parameters.

data_type

specifies the data type mapping for the corresponding column.

operator

specifies the property that corresponds to the query operator that you want to used to filter results. This value can be `#eq`, `#lt`, `#gt`, `#le`, `#ge`, `#ne`, or `#in`. See "Filtering and URI parameters" for details.

uri_property

specifies the name of the URI property to be filtered by the operator.

merge_behavior

specifies how the values of the column will be merged with the POST body. Valid values are:

- `json`: The value of the column is merged into the body as JSON.
- `replace`: The value of this column replaces all other POST parameters included in the POST body. This provides control over all of the POST body's parameters using a single set of properties.

default_parameter

(optional) specifies the name of the default parameter value to be filtered by the operator. If the default property is omitted, the value must be specified in the WHERE clause to filter by this column.

For example, the following demonstrates an entry for a POST request using an unparameterized request.

```
"countries2": {
          "#path": "http://example.com/country/",
          "#post": {
               "start_date":"2018-08-31",
               "end_date":"2018-09-01",
               "departments":"[engineering,marketing,sales]",
               "tags":"[blue,green,red]"
          }
     },
```

For example, the following demonstrates an entry for a POST request using a parameterized request.

```
"football": {
          "#path": "http://example.com/football/{team:Wildcats}",
          "#post": {
               "opponent":"Tigers",
               "date":"2018-2-2",
          }
     },
```

For example, the following demonstrates an entry for a POST request with parameters.

```
"incidents": {
          "#path": "https://www.example.com/safety/",
          "#post": {
               "departments":"accounting",
               "date":"2015-10-8",
               }
           "reported":{
                  "#type":"date",
                  "#eq":"reportedOn"
          }
     },
```

For example, the following demonstrates an entry for a POST request with custom parameters.

```
"incidents": {
          "#path": "https://www.example.com/issues/",
          "#post": {
               "id":"99-99999",
               "date":"2015-10-8",
               }
           "department":{
                  "#type":"VarChar",
                  "#postParameter":"json"
                  "#default": "{\"employee\": []}"
          }
     },
```

See also

Query paths on page 230

POST requests with an empty body

Typically, POST requests must pass a payload in the body element of the request; however, for certain services, there are endpoints that require POST requests to pass empty body elements to return results. For these endpoints, raw data is returned in the response of the request, instead of data that is dictated by parameters specified in the payload.

You can configure the driver to send POST request with an empty body by setting the `omitWhenEmpty` parameter to `false` in the POST body. In addition, you must set the media type for the response in the `content-type` header and, if the authentication information is different than the default for your REST model, authentication information for the endpoint (See "Custom authentication requests" for details).

Note that if you do not set the `omitWhenEmpty` parameter to `false` or set it `true` (default), the driver returns an error if the body is empty on a POST request. Conversely, if your POST request body includes a payload, the driver will ignore the setting of `omitWhenEmpty` parameter and issue a request with the contents of the body.

An entry for a POST request with an empty body takes the following form:

```
"<table_name>": {
          "#path": "<host_name>/<endpoint_path>",
          "#post": {
              "#omitWhenEmpty": false"
          },
          "#headers": {
            "content-type": "<content_type>"
          }
     },
```

table_name

is the name of the relational table to which the driver maps the endpoint. For example, `countries`.

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the Host Name (Hostname) option.

endpoint_path

is the path component of the URL endpoint. For example, `country`. This can be an unparameterized or parameterized path, a path that uses query parameters, or an array of paths. See "Query paths" for examples and more information.

content_type

is the media type used in the request and response.

For example, the following demonstrates an entry for a POST request with an empty body.

```
"countries": {
          "#path": "http://example.com/country/",
          "#post": {
             "#omitWhenEmpty": false
          },
          "#headers": {
             "content-type": "application/json"
          }
   },
```

See also

Custom authentication requests on page 206
Query paths on page 230

Requests with custom HTTP headers

Custom HTTP headers can be used to filter requests or specify a value for the Accept header.

Some endpoints employ custom HTTP headers to filter data returned by a GET or POST request. This type of filtering is typically used to create multiple unique reports/tables from the same endpoint. To use custom headers, you must define the request in the Model file. The Model file entry is comprised of a path and header object. The path object contains the URL endpoint used in requests, while the header object defines the headers and provides value arguments used to filter the request.

Additionally, the header object can be used to specify a value for the Accept header. This can be useful in the following scenarios:

- If a service returns responses in multiple supported formats (JSON, XLM, CSV). By default, the driver attempts to use JSON format when multiple formats are available; however, if you prefer, you can specify a different format using the Accept header. For example, to specify the XML format:

  ```
  "#headers":{
    "Accept": "application/xml"
  }
  ```

- If the default Accept header, `application/json`, is not accepted by the endpoint. This scenario typically occurs when accessing a vendor endpoint that uses a proprietary Accept header.

An entry for a GET request using custom HTTP headers takes the following form for an entry that defines three headers. You can define one or more headers in an entry.

```
"<table_name>":{
      "#path": "<host_name>/<endpoint_path>",
      "#headers":{
           "<header1>":"<value1>",
           "<header2>":"<value2>",
           "<header3>":"<value3>"
      }
}
```

table_name

is the name of the relational table to which the driver maps the endpoint. For example, `people`.

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the ServerName property.

endpoint_path

is the path component of the URL endpoint. For example, `times`. This can be an unparameterized or parameterized path, a path that uses query parameters, or an array of paths. See "Query paths" for examples and more information.

header

is the HTTP header component of the *header=value* pair used for filtering the request. For example, `X-Subway-Payment`.

When overriding the Accept header, this value is `Accept`.

value

is the value argument for the HTTP header used for filtering the request or, if overriding the default Accept header, the value of the Accept header for the endpoint. For example, `token`.

For example, the following demonstrates an entry for a GET request that defines custom HTTP headers.

```
"people":{
     "#path": "http://example.com/people",
     "#headers":{
          "Accept":"application/calendar+json",
          "X-Subway-Payment":"token",
          "X-Laundry-Service":"dryclean",
          "X-Favorite-Food":"pizza"
      }
},
```

See also

Query paths on page 230

Requests with custom parameters

Some services support custom parameters that the driver can leverage to process more efficient queries when filtering results. You can specify custom parameters using the `#queryParameter` in the column definition.

In addition, the driver supports filtering by parameters of custom languages, such as Jira Query Language (JQL) or Salesforce Object Query Language (SOQL). When the service supports the language specified by the `#queryParameter` property, the driver passes the query to the service to filter the results based on the parameter value specified by the `#default` property or by the Where clause in a SQL statement. The service then processes the results before returning them to the driver, thereby reducing the number of web service calls and driver overhead.

```
"<table_name>": {
          "#path": ["<host_name>/<endpoint_path>"]
          "<column_name1>":"<data_type1>",
          "<column_name2>":{
               "#type":"<data_type2>",
               "#queryParameter":"<param_name>",
               "#default":"<default_param>"
          }
     },
```

table_name

is the name of the relational table to which the driver maps the endpoint. For example, `people`.

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the Host Name (HostName) option.

endpoint_path

is the path component of the URL endpoint. For example, `times`. This can be an unparameterized or parameterized path, a path that uses query parameters, or an array of paths. See "Query paths" for examples and more information.

column_name

is the name of the column name that contains the nested object.

data_type

specifies the data type mapping for the column.

param_name

specifies the name of the query parameter that would appear in the request. This can be used for sending the value as a custom language. For example, `JQL` or `SOQL`.

default_param

(optional) specifies the argument parameter component of the *`parameter=value`* pair used for filtering the request. If the `#default` property is omitted, the value must be specified in the Where clause when issuing a query.

For example, the following demonstrates an entry for a GET request that defines custom parameters.

```
"issues": {
           "#path": ["http://example.org/issues"]
           "id":"VarChar",
           "query":{
                "#type":"VarChar",
                "#queryParameter":"jql",
                "#default":"FILTER ON NAME"
           }
      },
```

See also

Requests with query parameters on page 233
Query paths on page 230

Query paths

The query path is the endpoint(s) against which requests are issued. The path can be specified as a single endpoint or an array of endpoints (see "Array of endpoints" for details). You can specify the endpoints as a table name-endpoint pair (`"<table_name>":"<endpoint>"`) or by using the `#path` property in a table definition. The following types of paths are supported:

- Unparametrized paths
- Parametrized paths
- Paths with query parameters

By default, query paths are issued as GET requests unless they are specified in a POST entry. See "POST requests" for details.

The basic syntax of a query path takes the following form:

```
"<host_name>/<endpoint_path> <json_root>"
```

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the Host Name (HostName) connection option.

endpoint_path

is the path component of the URL endpoint. The value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.

json_root

(optional) is a simple path to the element containing the results. If the results are returned in a top-level array, nothing needs to be stated. For nested elements, separate the element names with forward slashes (/).

For example, the following demonstrates a query path for an unparamaterized GET request with a JSON root of `countries`.

```
#path:"http://example.com/countries/ countries",
```

See also

POST requests on page 224
URL-encoded values on page 264

Requests with unparameterized paths

Unparametrized requests are issued as GET requests, unless they are specified in a POST request entry. To specify endpoints for unparameterized requests, use the following format:

```
"<host_name>/<endpoint_path>",
```

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the Host Name (HostName) option.

endpoint_path

is the path component of the URL endpoint. For example, `countries`. The value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.

For example, the following demonstrates a GET request that will map to the `countries` table using the `#path` property.

```
#path:"http://example.com/countries/",
```

See also

URL-encoded values on page 264
POST requests on page 224

Requests with parameterized paths

Parameterized requests are issued as GET requests, unless they are specified in a POST request entry. To specify parameterized requests, use the following format:

```
"<host_name>/<endpoint_path1>/{<param_name>:<param_value>}[/<endpoint_path2>]",
```

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the Host Name (HostName) option.

endpoint_path

is the path component of the URL endpoint. For example, `states`. The value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.

param_name

is the parameter identifier used for filtering the request. For example, `countryCode`.

param_value

is the parameter value used for filtering the request during sampling. For example, `USA`.

For example, the following demonstrates a GET request that will map to the `states` table.

```
#path:"http://example.com/states/get/{countryCode:USA}/all",
```

See also

URL-encoded values on page 264
POST requests on page 224

Requests with query parameters

Requests with query parameters are issued as GET requests, unless they are specified in a POST request entry. Use the following format to specify endpoints for requests with argument parameters. Multiple argument parameters within the same endpoint are separated by an ampersand (&).

Note: For POST requests, the query parameter is specified in the body of the entry. See "Post requests" for details.

```
"<host_name>/<endpoint_path>?<parameter>=<value>[&...]",
```

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the Host Name (HostName) option.

endpoint_path

is the path component of the URL endpoint. For example, `times`. The value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.

parameter

is the argument parameter component of the *parameter=value* pair used for filtering the request. For example, `interval`.

value

is the value argument parameter used for filtering the request. For example, `5min`.

For example, the following demonstrates a GET request that will map to the `timeseries` table.

```
#path:"https://www.example.com/times/query?interval=5min&symbol=USA&function=TIME_SERIES_WEEKLY",
```

See also

URL-encoded values on page 264
POST requests on page 224

Arrays of endpoints

You can specify an array of endpoints in a comma-separated list using the `#path` property. This allows you to specify multiple endpoints to different representations of the same data. When a query is executed, the driver maximizes performance by determining which endpoint would return the smallest result set that satisfies your query; then, issues a request to that endpoint. Arrays of endpoints are issued as GET requests, unless they are specified in a POST request entry.

Important: To determine the endpoint best suited for your query, starting at the top of the array, the driver attempts to match the WHERE clause parameter to the supplied paths. The driver will use the first endpoint in the list that successfully satisfies the query; therefore, the endpoints should be specified in an order of most-specific to least-specific to ensure that the most appropriate endpoint is used.

The following demonstrates the basic syntax for issuing an array of endpoints. Using this form, you may specify two or more endpoints in an array.

```
#path:"[
   <host_name>/<endpoint_path1>,
   <host_name>/<endpoint_path2>,
   <host_name>/<endpoint_path3>
]
```

host_name

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the Host Name (HostName) option.

endpoint_path

is the path component of the URL endpoint. For example, `times`. The value must be URL-encoded using valid syntax. For example, spaces in an endpoint are replaced with `%20`. See "URL-encoded values" for details.

The following demonstrates an array of endpoints. In this example, `/orders/{orderid}` returns data for just one order, `/customer/{custid}/orders` returns data for all orders for a customer, and `/orders` returns all orders. Note that the array is specified in order from most-specific to least-specific to ensure that the driver uses the endpoint best suited for your query.

```
{
   "Orders":{
      #path:"[
          "/orders/{orderid}",
          "/customer/{custid}/orders",
          "/orders"
   ]
}
```

For example, if you executed the following query, the driver would return results for order `abc123` from the `/orders/{orderid}` end point.

```
SELECT * FROM ORDERS WHERE ORDERID=abc123
```

The following query would return all the results for the customer with an ID of `98765` from the `/customer/{custid}/orders` endpoint.

```
SELECT * FROM ORDERS WHERE CUSTID=98765
```

The following query would return results for all orders from the `/orders` endpoint.

```
SELECT * FROM ORDERS
```

See also

URL-encoded values on page 264

Column names

The column name specified in a table definition can be the element name of the JSON response or a regular expression matching an element in the JSON response.

Regular expressions (Java Regex)

When specifying a regular expression, the name should begin with a tilde (~). For example, if you had a parameter that returned a JSON field as `Time Series (Daily)` or `Weekly Time Series`, you could specify a regular expression `~.*Time Series.*` for the column name. This would cause the column to be reported as `TIMESERIES`, regardless of the contents of the field. For more information on Java Regex syntax, refer to the Java documentation.

Aliases

You can also specify alias column names by specifying the alias name in angle brackets (`< >`) after the column name. This is useful if the generated column name is confusing or lacks a real world context. For example:

```
"userfield73<casenumber>": "varchar(10)"
```

Data type mapping

You can manually configure the mapping of data types to a column using the following syntax in a column definition.

```
"<column_name>":"<data_type>(<size_parameters>)",
```

column_name

is the name of the column in your relational table.

data_type

is the case-insensitive name of the data type to which you want to map the column. For columns for which no data type is defined, the driver heuristically maps the column to the most appropriate data type. If a value of `null` is specified, the column is mapped to the default data type, `varchar(50)`. See the "Supported Data Types and Parameters" table for a list supported data types.

size_parameters

(optional) is the length, precision and/or scale of the specified data type. If this value is not specified, the driver will use the default value for the data type.

For example:

```
"price":"decimal(18.2)",

"name":"Varchar(256)",

"age":"Integer",
```

The following table documents the supported data types and parameters.

Table 38: Supported data types

Data Type and Parameters	Characteristics
BigInt	Range: $-99*10^{18}$ to $99*10^{18}$
Binary(*l*)	Range: $-99*10^{32765}$ to $99*10^{32765}$
Bit	Valid values: `0` or `1`
Boolean	Valid values: `0` or `1`
Char(*l*)	Precision: `255`
Date[2]	Precision: `10`
Decimal(*p.s*)	Range: $-99*10^{14}$ to $99*10^{14}$ Minimum scale: `0` Maximum scale: `32767`
Double	Range: $-99*10^{51}$ to $99*10^{51}$
Float	Range: $-99*10^{22}$ to $99*10^{22}$
GUID	Range: $-99*10^{522}$ to $99*10^{522}$
Integer	Range: $-99*10^{8}$ to $99*10^{8}$
JSON	Precision: `16777215`
LongVarBinary(*l*)	Precision `16777215`
LongVarChar(*l*)	Precision: `16777215`
NVarChar(*l*)	Precision: `32767`
SmallInt	Range: `-99999` to `99999`
Time(*s*)[2]	Precision: `12` Minimum scale: `0` Maximum scale: `9`
Timestamp(s)[2]	Precision: `23` Minimum scale: `0` Maximum scale: `9`
TinyInt	Range: `-999` to `999`

[2] Formats for Data, Time, and Timestamp values are configurable. For more information, see Date, time, and timestamp formats on page 240.

Data Type and Parameters	Characteristics
VarBinary(*l*)	Precision: `16777215`
VarChar(*l*)	Precision: `32767`

Primary key

You can designate the primary key for a table by modifying the Model file. In the column object, add the `#key` after the data type element, separated by a comma.

Note: When designating a new primary key, you can query the `INFORMATION_SCHEMA.SYSTEM_SAMPLING_STATUS` system table for a list of potential primary key candidates. See "Determining the primary key" for more information.

In the following example, the `employeeID` column has been designated the primary key for this table.

```
{
"my_table":{
            "#path":[
                "https://example.com/employees"
           ],
           "employeeID":"VarChar(32),#key",
           "position_title":"VarChar(46)",
           "start_year":"Integer",
            }
}
```

You an also create a composite primary key by using the `#key` element to designate multiple columns in a definition. For example, the values of the `employeeID` and `position` columns act as a composite key in the following:

```
{
"my_table":{
            "#path":[
                "https://example.com/employees"
           ],
           "employeeID":"VarChar(32),#key",
           "position":"Integer",#key,
           "position_title":"VarChar(46)",
           "start_year":"Integer",
            }
}
```

See also

Determining the primary key on page 48

Read-only columns

You can designate columns as read-only, which overrides any write operations enabled for the table. To flag a column as read-only, add the `#readOnly` element after the data type element, separated by a comma. Note that columns marked with `#key` (primary key) element are read-only by default.

In the following example, the `postion_title` column has been designated as read-only, and the `employeeID` column is read-only because it has been designated the primary key.

```
{
"my_table":{
             "#path":[
                 "https://example.com/employees"
            ],
            "employeeID":"VarChar(32),#key",
            "position_title":"VarChar(46),#readOnly"
            "start_year":"Integer",
             }
}
```

See also

Write operations on page 213

Nullable columns

You can flag columns as nullable for metadata purposes. To flag a column as nullable, specify the `#notNull` element after the data type element, separated by a comma. Note that columns marked with `#key` (primary key) element are read-only by default.

In the following example, the `postion_title` column has been designated as nullable, and the `employeeID` column is nullable because it has been designated the primary key.

```
{
"my_table":{
             "#path":[
                 "https://example.com/employees"
            ],
            "employeeID":"VarChar(32),#key",
            "position_title":"VarChar(46),#notNull"
            "start_year":"Integer",
             }
}
```

Columns as an array

A column is defined as an array by ending the column name in brackets (`[]`). When mapping a column with an array to the relational model, the driver normalizes the column to a child table. The driver also supports arrays nested in arrays. When generating the relational view, the driver normalizes the nested array to child table.

A column as an array takes the following form for defining two nested columns. You can define one or more nested columns in an array.

```
"<column>[]":{"<array_column_a>":"<data_type>","<array_column_b>":"<data_type>"}
```

column_name

is the name of the column name that contains the nested object.

array_column

specifies the name of the column in an array.

data_type

specifies the data type mapping for the corresponding column.

For example:

```
"income[]":{"month":"string","amount":"decimal(18.2)"}
```

Columns as a key-value map

You can define a column as an key-value map by ending the column name in curly brackets ({ }), followed by an object enclosed in curly brackets ({ }). In addition, you can specify the data type for the key in the curly brackets in the column name. For date and time data types, if necessary, you can also specify a format after the key data type and separated by a comma.

A column as an key-value map takes the following form for a key-value map with two nested columns. You can define one or more nested columns in a key-map entry.

```
"<column_map>{<key_data_type>,<format>}":{"<column_a>:<data_type>","<column_b>":"<data_type>"
```

column_map

is the column name that contains the key-value map.

key_data_type

specifies the data type mapping for the map key.

format

(optional) specifies the Java SimpleDateFormat, if the data type is of the data, time, timestamp type. This value is not required if the values use ISO format. See "Date, time, and timestamp formats" for details.

column

specifies the name of columns within the key-value map.

data_type

specifies the data type mapping for the column within the key-map.

The following example demonstrates defining a column as a key-value map with the key data type and format specified:

```
"balancesheet{Date,MMddyyyy}":{"assets":"decimal(18.2)", "liabilities":"decimal(18.2)"}
```

See also

Date, time, and timestamp formats on page 240

Columns with nested objects

The driver supports objects with nested objects. When generating the relational view, the driver flattens nested objects to the table of the parent object.

A column with three nested columns takes the following form for a column with three nested objects. You can define one or more nested objects in a column definition.

```
"<column_name>":{
    "<nested_column1>":"<data_type>",
    "<nested_column2>","<data_type>",
    "<nested_column3>","<data_type>"
   }
```

column_name

is the name of the column that contains the nested object.

nested_column

specifies the data type mapping for the map key.

data_type

specifies the data type mapping for the corresponding column.

The following example demonstrates defining a column with nested objects:

```
"address":{"line1":"varchar(256)", "zip":"integer", "city":"varchar(256)"}
```

Date, time, and timestamp formats

By default, the driver interprets values of the Data, Time, and Timestamp data types using the default ISO 8061 formats:

- `YYYY-MM-DD`
- `HH:MM:SS.sssssssssZ`
- `YYYY-MM-DDTHH:MM:SS.sssssssssZ`

The driver provides additional flexibility in parsing ISO formats:

- The value can consist of less than the full number of digits. For example, `1970-1-1` is acceptable, as opposed to `1970-01-01`.
- The fractional second and timezone values are optional.
- For timestamps, dates or the date portions of values can use `/` or `-` as separators.
- For timestamps, the separator between the date and time portions can be an empty space instead of a `T`.

However, if necessary, you can also specify your own format after the data type element (or after `#key` element in the primary key column) in your column definition, using the Java SimpleDateFormat. These definitions take the following form:

```
"<column_name>":"<data_type>","<java_date_format>"
```

where:

column_name

is the name of the column.

`data_type`

is either the `Date`, `Time` or `Timestamp` data type.

`java_date_format`

specifies the format of your Date, Time, or Timestamp values using the Java SimpleDateFormat. For more information on the Java SimpleDateFormat, refer to https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html.

For example:

```
"birthday":"date","d-M-y-G"
```

Subfields

Sometimes when a value comes back as a string, only part of that string is required. The `#extract` property allows you to specify a regular expression that returns only a portion of the string. In addtion, using the `#type` property, you can map the appropriate data type for the subfield before it is converted to the local type.

A column that extracts a subfield takes the following form:

```
"<column_name>": {"#type":"<data_type>","#extract:<reg_expression>"}
```

For example, suppose you get back a color value as `27:red:#ff0000`, but you only need to know that it is color `27` . You can accomplish this by specifying the following definition:

```
"color":{"#type":"Integer","#extract":"^([0-9]+).*"}
```

This results in the driver returning only the numeric portion of the string, which will be converted into an integer.

Columns as HTTP headers

A column can be sent as an HTTP header instead of as part of a query string in GET requests. HTTP headers can be specified in the column definition by setting the `#header` property to `true` and providing the header using the `#eq` property.

If the header to be filtered is a dynamic HTTP-request header, and therefore not returned in the result set, specify `#virtual:true` to have it exposed as searchable column. Otherwise, this property should be omitted.

Note: The driver does not support headers that are natively mapped to the JSON data type to be exposed as virtual columns. If the header is mapped to a JSON data type, the driver does not treat the header field as a virtual column and looks for it in the response. You can avoid this issue by natively mapping virtual fields that contain JSON strings to the VarChar data type.

The syntax to send a column as an HTTP header takes the following form:

```
"<column_name>[]":{
   "#type":"<data_type>",
   "#header":true,
   "#virtual":true,
   "#default":"<default_filter>",
   "#eq":"<header_name>"
}
```

where:

data_type

(optional) specifies the data type to which the column is mapped.

default_filter

(optional) specifies the value sent with the header that is used to filter results. When this value is specified, the value `<header>:<default_filter>` is sent in the HTTP request. If the default filter is not specified, a WHERE clause must provide the filter value. For example:

```
SELECT * FROM authentication WHERE authentication = 'scott/tiger'
```

header_name

specifies the name of the HTTP header sent in the request.

The following example demonstrates

```
"service":{"#type":"VarChar","#header":true,"#default":"scott/tiger","#eq":"X-Custom-Auth"}
```

Filtering and URI parameters

The Model file supports a number of query operators that can be used to filter results. When specifying the operators in column definition or URI, the filtering is pushed down to the data source, instead of being handled by the driver. This results in more efficient processing of queries and improved performance. You can specify one or more operators in the column definition using the set of Model file properties in the "Query operator syntax" table.

If the URI property to be filtered is a parameter for the URI or POST body, and therefore not returned in the result set, specify `#virtual:true` to have it exposed as searchable column. Otherwise, this property should be omitted.

Note: The driver does not support URI properties that are natively mapped to the JSON data type to be exposed as virtual columns. If the URI property is mapped to a JSON data type, the driver does not treat the URI property field as a virtual column and looks for it in the response. You can avoid this issue by natively mapping virtual fields that contain JSON strings to the VarChar data type.

The syntax to send a column as using operators takes the following form:

```
"<column_name>":{
   "#type":"<data_type>",
   "<operator>":"<uri_parameter>",
   "#default":"<default_parameter>",
   "#virtual":true
}
```

where:

data_type

specifies the data type to which the column is mapped.

Note: If the data type is a date, time, timestamp, you can determine the format used by specfiying a Java SimpleDateFormat string after a comma. See "Date, time, and timestamp formats" for details.

---

operator

specifies the property that corresponds to the query operator that you want to used to filter results. This value can be #eq, #lt, #gt, #le, #ge, #ne, or #in. See "Query operator syntax" table for details.

uri_property

specifies the name of the URI property to be filtered by the operator.

default_param

(optional) specifies the default parameter when the URI property to be filtered is a parameter. Some REST services require certain parameters in order to operate. Typically, this would require including a WHERE <parameter>=<value> in a SQL statement. However, when specifying the default parameter, the driver will push down this value when it's not included in the statement.

Table 39: Query operator syntax

Query Operator	Property syntax
=	"#eq":"<uri_property>"
<	"#lt":"<uri_property>"
>	"#gt":"<uri_property>"
!=	"#ne":"<uri_property>"
>=	"#ge":"<uri_property>"
<=	"#le":"<uri_property>"
IN	"#in":"<uri_property>"

Examples

The following demonstrates an entry using filters for the orderdate column.

```
{
    "Orders":{
        #path:"[
            "/orders/{orderid}",
            "/customer/{custid}/orders",
            "/orders"
        ],
        "orderid":"Varchar(256)",
        "custid":"Varchar(256)",
        "orderdate":{
           "#type":"Date",
           "#eq":"date",
           "#gt":"after",
           "#lt":"before"
        }
    }
}
```

The following demonstrates example queries to use against the preceding entry along with corresponding example URIs that can be issued as an alternative to specifying filters in the column definition.

- The following query returns results for all the orders that occurred on 2020-01-01:

  ```
  SELECT * FROM ORDERS WHERE ORDERDATE = '2020-01-01'
  ```

  Instead of using the column definition, you can also push down filtering for this query using the following URI:

  ```
  https://www.example.com/ORDERS?DATE=2020-01-1
  ```

- The following query returns all the orders that occurred after 2020-01-01:

  ```
  SELECT * FROM ORDERS WHERE ORDERDATE > '2020-01-01'
  ```

  Instead of using the column definition, you can also push down filtering for this query using the following URI:

  ```
  https://www.example.com/ORDERS?AFTER=2020-01-01
  ```

- The following query returns results for all the orders that occurred before 2020-01-01:

  ```
  SELECT * FROM ORDERS WHERE ORDEREDATE < '2020-01-01'
  ```

  Instead of using the column definition, you can also push down filtering for this query using the following URI:

  ```
  https://www.example.com/ORDERS?BEFORE=2020-01-01
  ```

See also

Date, time, and timestamp formats on page 240

User-defined functions and procedures

The driver supports user-defined procedures and functions that are defined by the application or in the Model file. Functions and procedures are logically grouped statements that can be used to access data in the REST service or call additional functions and procedures. Since functions and procedures are saved, you can use them to store commonly used code and call them whenever needed, which can improve memory usage, initial coding effort, and coding maintenance.

Note that when the application defines a function or procedure, the driver stores the definition in internal memory for only the life of the session. Therefore, stored procedures and functions must be defined in a session before they are called. If you wish for stored procedures or functions to persist between sessions, use the Model file to store your definitions.

This section describes the syntax used to define functions and procedures supported by the driver. The syntax for defining the function in the application and the Model file is identical, with the exception of the `#routines` entry tags being exclusive to the Model file. For an overview of the syntax, see:

- Function syntax
- Procedure syntax

Function syntax

The following demonstrates the basic syntax used when defining a function. Note that not all the clauses in this example are required. See the following sections for more information on these statements and supported syntax not defined in this example.

Note: The `#routines` entry tags are used only in the Model file, not when defining a function with the application.

```
"#routines":[
CREATE FUNCTION <function_name> (<parameters>) RETURNS <data_type>(<p>,<s>)[,...]
    [NOT] DETERMINISTIC
    [RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT]
    <language>
    SPECIFIC <reference_name>
    //For SQL, use the RETURN keyword for single-line statements. For compound
    //statements, use the Body keyword. Neither keyword is used for Java functions.
    [RETURN | BEGIN ATOMIC <routine_body>]
    END;"
]
```

`function_name`

Specifies the name of the function to be called by the application.

`data_type`

Specifies the data type, including the precision and scale, for the results of the function. See "Data types syntax" for a complete list of supported data types. See Data types syntax on page 248 for more information and supported syntax.

`parameters`

Defines the parameter variables used in the function. See Parameters on page 247 for more information and supported syntax.

`language`

Specifies the language used by the routine. For example, if the function only reads data from the underlying data source, you would specify the `READS SQL DATA` keywords. See Routine language on page 250 for more information and supported syntax.

`reference_name`

Optionally, specifies the unique reference name to be assigned for polymorphic functions. To access polymorphic functions through the DDL, each function is assigned a generated implementation name for reference in schema manipulation commands. To specify a name for these functions, instead of a generated one, use the `SPECIFIC` keyword. See SPECIFIC statements on page 249 for more information and supported syntax.

`routine_body`

Specifies the statements used to perform the operation of the function. This can be a single statement defined by the `RETURN` statement or multiple statements wrapped by the `BEGIN ATOMIC` and `END` keywords. See Compound statements on page 252 for more information and supported syntax.

Procedure syntax

The following demonstrates the basic syntax used when defining a procedure. Note that not all the clauses in this example are required. See the following sections for more information on these statements and supported syntax not defined in this example.

Note: The `#routines` entry tags are used only in the Model file, not when defining a procedure with the application.

```
"#routines":[
CREATE PROCEDURE <procedure_name> (<parameters>) RETURNS <data_type>(<p>,<s>)[,...]
    [NOT] DETERMINISTIC
    <language>
    SPECIFIC <name>
    DYNAMIC RESULT SETS <sets_returned>
    //For SQL, use the RETURN statement for single-line statements. For compound
    //statements, use BEGIN ATOMIC. Neither keyword is used for Java procedures.
    [RETURN | BEGIN ATOMIC <routine_body>]
    END;"
]
```

`procedure_name`

Specifies the name of the procedure to be called by the application.

`data_type`

Specifies the data type, including the precision and scale, for the results of the function. See Data types syntax on page 248 for more information and supported syntax.

`parameters`

Defines the parameter variables used in the procedure. See Parameters on page 247 for more information and supported syntax.

`language`

Specifies the language used by the routine. For example, if the function only reads data from the underlying data source, you would specify the `READS SQL DATA` keywords. See Routine language on page 250 for more information and supported syntax.

`reference_name`

Optionally, specifies the unique reference name to be assigned for polymorphic functions. To access polymorphic functions through the DDL, each function is assigned a generated implementation name for reference in schema manipulation commands. To specify a name for these functions, instead of a generated one, use the `SPECIFIC` keyword. See SPECIFIC statements on page 249 for more information and supported syntax.

`routine_body`

Specifies the statements used to perform the operation of the procedure. This can be a single statement defined by the `RETURN` statement or multiple statements wrapped by the `BEGIN ATOMIC` and `END` statements. In addition to DQL, DML, and DDL statements, the driver supports a number of procedural SQL statements. See Compound statements on page 252 for more information and supported syntax.

Parameters

Parameters

You can define one or more parameters in your function or procedure using the following syntax. When specifying multiple parameters, you must separate parameters with a comma.

Functions syntax:

```
([IN] name <data_type>[,...])
```

Procedures syntax:

```
([IN | OUT | INOUT] <data_type> [,...])
```

For procedures, you can specify `IN`, `OUT`, or `INOUT` parameters. The default is `IN`.

Not that both `OUT` and `INOUT` parameters must be registered in the procedure to be usable.

The following examples demonstrate using parameters.

Example 1:

```
CREATE PROCEDURE swap(INOUT i BIGINT, INOUT j BIGINT)
    BEGIN ATOMIC
    DECLARE k BIGINT;
    SET k = i;
    SET i = j;
    SET j = k;
END
```

Example 2:

```
try (CallableStatement cs = c.prepareCall("CALL swap(?, ?)")) {
    cs.registerOutParameter(1, Types.SQL_BIGINT);
    cs.registerOutParameter(2, Types.SQL_BIGINT);
    cs.setLong(1, value1);
    cs.setLong(2, value2);
    cs.execute();
    Assert.assertEquals(cs.getLong(1), value2);
    Assert.assertEquals(cs.getLong(2), value1);
}
```

Data types syntax

The following demonstrates the supported syntax for data types supported in functions and procedures. Note that any data type supported by the driver is supported in a function or procedure. See "Data types" for more information on data types supported by the driver.

```
BigInt |
Binary(ll)|
Boolean |
Char(ll)|
Date |
Decimal(pp, ss)|
Double |
Float |
GUID |
Integer |
JSON |
LongVarBinary(ll)|
LongVarChar(ll)|
NVarChar(ll)|
SmallInt |
Time((t))|
Timestamp((t))|
TinyInt|
VarBinary(ll)|
VarChar(ll)
```

where:

`ll`

is a value from `1` to `99` that specifies the length of the field.

`pp`

is a value from `1` to `99` that specifies the precision of the field.

`ss`

is a value from `1` to `99` that specifies the scale of the field.

`t`

is a value from `1` to `9` number of digits in the fractional seconds value of the field.

See also

Data types on page 38

DETERMINISTIC statements

The `DETERMINIST` statement determines whether the return value is based solely on the input values or if they could depend on additional, varying data.

```
[NOT] DETERMINISTIC
```

where:

`DETERMINISTIC`

specifies that the return value or operation is dependent solely on the input values. Specify this statement if the same input values always generate the same output.

`NOT DETERMINISTIC`

specifies that the return value of the function could depend on variables, such as data read during the evaluation, random values, or the current time. If the same input parameters may not always generate the same output, specify this statement or rely on the default behavior.

Note that this statement is optional. The default is `NOT DETERMINISTIC`.

NULL-input statements

Null-input statements determine how NULL parameters in functions are handled by the SQL Engine.

```
RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT
```

where

`RETURNS NULL ON NULL INPUT`

Specifies that the function returns NULL if any of the parameters are NULL. This statement is intended to allow the SQL Engine to skip evaluation and imply the results as NULL, which can improve performance and simplify logic.

`CALLED ON NULL INPUT`

Specifies that the SQL Engine evaluates the function even when one of the parameters is NULL.

This statement is not required. The default is `CALLED ON NULL INPUT`.

SPECIFIC statements

The driver supports polymorphic functions in SQL. To access those functions through the DDL, each function or procedure is assigned a generated implementation name to make it unique for reference in schema manipulation commands. If you prefer to provide your own reference name, instead of the generated one, you can specify it using the SPECIFIC statement. Using this statement assists in referencing the correct function in metadata or DDL statements when you have multiple similar functions.

```
SPECIFIC <reference_name>
```

where:

`reference_name`

is a unique reference created by you to be used in schema manipulation commands.

Routine language

The following language statements specify whether executing the function will change the state of the database. The default is CONTAINS SQL.

Functions syntax:

```
NO SQL | CONTAINS SQL | READS SQL DATA
```

Procedures syntax:

```
NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA
```

where:

NO SQL

specifies that the routines use Java language.

CONTAINS SQL

specifies that routines use SQL statements, but does not directly access the underlying data source.

READS SQL DATA

specifies that routines use SQL statements to access the underlying data source, but only reads the data.

MODIFIES SQL DATA

specifies that routines use SQL statements to access and potentially alter the underlying data source.

EXTERNAL statements

The EXTERNAL statement specifies the name and classpath of external methods used by the routine. Note that the Java method must be a public static method in the public class. To specify an EXTERNAL statement, use the following syntax:

```
EXTERNAL <method_name> CLASSPATH: <classpath>
```

where

method_name

is the name of a method external to the function or procedure to be called by the routine.

classpath

is the name of the classpath for your external method.

Result sets

In addition to defining output parameters, procedures can return one or more result sets. When defining a procedure that returns a result sets, you must define the maximum number of result sets to be returned using the following syntax:

```
DYNAMIC RESULT SETS <xx>
```

`xx`

is the maximum number of result sets that could be returned by the procedure. The valid values are from `1` to `99`.

The default is that the procedure does not return any result sets.

RETURN statements

A function can have a single-line body that is comprised of a `RETURN` statement. Alternatively, multiple statements (compound statements) can be specified inside the `BEGIN ATOMIC` and `END` statements. Note that there can be more than one `RETURN` statement in the body. The following syntax is used for a `RETURN` statement:

```
RETURN <expression>
```

where

`expression`

is the argument that determines the value to be returned.

For example, the following function demonstrates using a RETURN statement in a compound statement:

```
// This function returns a time one hour before the event starts
RETURNS TIMESTAMP
BEGIN ATOMIC
    DECLARE max_event TIMESTAMP;
    SET max_event = SELECT MAX(start_time) FROM events WHERE type = e_type;
    RETURN max_event - 1 HOUR;
END
```

The next example returns same results as the prior without using the compound statements body:

```
CREATE FUNCTION an_hour_before_max (e_type INTEGER)
RETURNS TIMESTAMP
    RETURN (SELECT MAX(start_time) FROM events WHERE type = e_type) - 1 HOUR
```

The following example demonstrates a function that uses the `RETURN TABLE` syntax to return a table:

```
CREATE FUNCTION alice_and_friends(ignore INTEGER)
RETURNS TABLE(id INTEGER, name VARCHAR(32))
READS SQL DATA
BEGIN ATOMIC
//This function returns a table value. The value can be used by the caller
//anywhere a TABLE clause would go, like SELECT * FROM alice_and_friends(3);
    DECLARE TABLE temptable (id INTEGER, name VARCHAR(32));
    INSERT INTO temptable VALUES (1, 'Alice');
    INSERT INTO temptable VALUES (2, 'Bob');
    INSERT INTO temptable VALUES (3, 'Chuck');
```

```
    RETURN TABLE(SELECT * FROM temptable WHERE id != ignore);
END;
```

Compound statements

In addition to defining as single statement in a function using a `RETURN`, you can specify compound statements (multiple statements) wrapped in `BEGIN ATOMIC` and `END`. The following demonstrates the syntax along with supported typical declarations and statements. See the following topics for more information about supported statements and syntax.

```
BEGIN ATOMIC
         <table_variable_declaration>;
         <scalar_variable_declaration>;
         <cursor_declaration>;
         <handler_declaration>;
         <procedural_statement>;
         RETURN <expression>;
         END
```

where:

`table-variable-declaration`

(optional) defines the variable name, column names, and data types for a temporary table to be created. For example, `DECLARE TABLE mytable (id INTEGER, employees VARCHAR(32));`. See Table declarations on page 263 for more information.

`scalar_variable_declaration`

(optional) defines the variable name and data type for a value to be stored. For example, `DECLARE max_event TIMESTAMP;`. See Scalar declarations on page 262 for more information.

`cursor_declaration`

(optional) defines the cursor that allows procedures to return table values as result sets. See "Cursor declaration" for details and syntax. See Cursor declarations on page 255 for more information.

`handler_declaration`

(optional) defines a handler used for exception handling. See Handler declarations on page 257 for more information.

`procedural_statement`

defines DQL, DML, DDL, and SQL procedural statements used to execute the routine. See Procedural SQL statements on page 253 for more information.

`expression`

is the argument that determines the value to be returned.

Procedural SQL statements

In addition to regular DQL, DML, and DDL statements, the driver also supports a number of procedural SQL statements within function and procedure bodies. The following topics describe the supported procedural SQL statements and syntax. For more information on the syntax of procure bodies, see "Compound statements."

See also

Compound statements on page 252

CALL statements

The `CALL` statement is used to call a procedure. Functions and procedures can call procedures, even recursively. However, called procedures cannot return table values when called from inside another function or procedure. Only functions can return table values. Note that the top-level procedure can return values through the JDBC to the calling application.

```
CALL <procedure_name> (IN | OUT | INOUT <variable_name>[, ...])
```

`procedure_name`

is the name of the procedure you want to call.

`IN`

specifies that the parameter is an input parameter.

`OUT`

specifies that the parameter is an output parameter.

`INOUT`

specifies that the parameter is an input and output parameter.

`variable_name`

is the name of the variable in the parameter.

CASE statements

The `CASE` statement evaluates specified conditions and returns results once a condition has been met. There are two versions of the statement. One version uses a single expression that is compared to the values of each case, similar to switch statement in C-like languages and Java. The other version while the other compares the expression at each `WHEN` clause.

Note: If no `ELSE` clause is specified, an exception will be thrown if no conditions are met.

The following syntax demonstrates a statement with a single expression:

```
CASE <expression>
    WHEN <when_condition> THEN <procedural_statement>;
    [...;]
    ELSE <procedural_statement>;
END CASE
```

The following syntax demonstrates a statement that compares the expression at each `WHEN` clause:

```
CASE
    WHEN <boolean_expression> THEN <procedural_statement>;
    [...;]
    ELSE <procedural_statement>;
END CASE
```

where:

`expression`

is the expression to be evaluated against when conditions.

`when_condition`

is a condition that is compared to the expression.

`procedural_statement`

is the procedural statement that is executed when a condition is met.

`boolean_expression`

is a boolean expression used to determine whether the specified procedural statement is executed.

`ELSE`

specifies that if the preceding condition was not met, the specified statement should be executed.

The following examples produce the same results using the different versions of syntax.

Statement with a single expression:

```
CASE state
    WHEN 'create', 'insert' THEN INSERT INTO t_one ...;
    WHEN IN ('drop', 'delete', 'remove') THEN DELETE FROM t_one WHERE ...;
    WHEN IS NULL THEN SIGNAL 'HY000' SET MESSAGE = 'This program is befuddled';
    ELSE UPDATE t_one ...;
END CASE
```

Statement that compares the expression at each `WHEN` clause:

```
CASE
    WHEN state = 'create' OR state = 'insert' THEN INSERT INTO t_one ...;
    WHEN state IN ('drop', 'delete', 'remove') THEN DELETE FROM t_one WHERE ...;
    WHEN state IS NULL THEN SIGNAL 'HY000' SET MESSAGE = 'This program is
befuddled';
    ELSE UPDATE t_one ...;
END CASE
```

Column definitions

The following syntax allows you to define columns in various parts of the procedure, such as table variable declarations.

```
<column_name> <data_type>(<p>, <s>)
```

`procedure_name`

Specifies the name of the procedure to be called by the application.

`data_type`

Specifies the data type, including the precision and scale, for the results of the function. See "Data types syntax" for a complete list of supported data types.

See also

Table declarations on page 263

Cursor declarations

The following syntax declares the cursor that allows procedures to return table values as result sets. Declaring a cursor allows them to be used by the function or procedure; however, they are not part of the response until they are executed in an `OPEN` statement.

Note: There can be fewer `OPEN` statements in a procedure than stated by the `DYNAMIC RESULTS SETS` clause, but there cannot be more.

```
DECLARE cursor_name CURSOR WITH RETURN FOR <select_statement>
```

where:

`cursor_name`

specifies the name of the cursor that you are declaring.

`select_statement`

specifies the `SELECT` statement of which the cursor holds the results.

See also

OPEN statements on page 260

DELETE statements

The `DELETE` statement is used to delete rows from a table.

```
DELETE FROM <table_name> [WHERE <search_condition>]
```

where:

`table_name`

specifies the name of the table from which you want to delete rows.

`search_condition`

is an expression that identifies which rows to delete from the table.

Note: The `WHERE` clause determines which rows are to be deleted. Without a `WHERE` clause, all rows of the table are deleted, but the table is left intact. Where clauses can contain subqueries.

FOR statements

The `FOR` statement executes the specified `SELECT` statement and then iterates the specified procedural statement against each row in the result set.

```
[<label> :] FOR <select_statement> DO <procedural_statement> END FOR [<label>]
```

where:

`label`

(optional) is the identifier for the `FOR` statement that can be referenced by other statements. If specified, the label at the beginning of the statement must match the label at the end.

`select_statement`

is the `SELECT` statement used to return the result set.

`procedural_statement`

is the procedural statement to be iterated against each row in the result set.

Note:

- Column names in the `SELECT` list must not conflict with any other identifiers used as variables or parameters.
- Columns named in the statement are read-only for the duration of the `FOR` loop.
- The result list is built before the loop executes to prevent changes in the underlying data from changing the number of iterations in the loop.

For example:

```
CREATE PROCEDURE delete_duplicate_products()
MODIFIES SQL DATA
BEGIN ATOMIC
    DECLARE prev VARCHAR(32) DEFAULT '';
    FOR SELECT id,name FROM products DO
        IF product.name = prev THEN
            DELETE FROM products WHERE products.id = id;
        END IF;
        SET prev = name;
    END FOR;
END
```

Handler declarations

The following syntax is used to declare how error handlers handle exceptions when they occur.

```
DECLARE [UNDO | CONTINUE | EXIT] HANDLER FOR [SQLEXCEPTION | SQLWARNING | NOT FOUND
SQLSTATE <sql_state>[, ...]] [<procedural_statement>]
```

where:

`UNDO`

specifies that all the changes made inside the block are undone before executing the statement after the block, if any.

`CONTINUE`

specifies that the exception is ignored, then the next iteration or statement after the block, if any, is executed.

`EXIT`

specifies that the block is terminated, but any successful changes are preserved.

`SQLEXCEPTION`

specifies that the procedural statement is to be executed when a SQL exception is encountered.

`SQLWARNING`

specifies that procedural statement to be executed when a SQL warning is encountered.

`NOT FOUND`

specifies that the procedural statement is to be executed when a DELETE, UPDATE, INSERT, or MERGE statement is executed without affecting any rows.

`SQLSTATE`

specifies that the procedural statement is to be executed when the specified SQLSTATE code is encountered.

`sql_state`

specifies the five-character SQLSTATE code. For example, `HY000`.

`procedural_statement`

(optional) specifies the procedural statement to be executed when the conditions in the declaration are encountered.

IF statements

`IF` statements are used to execute statements if specified conditions are met.

```
IF <boolean_expression> THEN <procedural_statement>;
  [ELSEIF <boolean_expression> THEN <procedural_statement>;]
  [ELSE <procedural_statement>;]
END IF
```

where:

`boolean_expression`

is a boolean expression that is compared against a value.

`procedural_statement`

is the procedural statement that is executed when a condition is met.

`ELSEIF`

specifies that if the preceding condition was not met, the following conditions should be evaluated.

`ELSE`

specifies that if the preceding condition was not met, the specified statement should be executed.

INSERT statements

The Insert statement is used to add rows to a table.

```
INSERT INTO <table_name> [(<column_name>[, ...)] {VALUES (expression [, ...]) |
<select_statement>}
```

where:

`table_name`

is the name of the table in which you want to insert rows.

`column_name`

is optional and specifies an existing column.

`expression`

is the list of expressions that provides the values for the columns of the new record.

`select_statement`

is a query that returns values for each column_name value specified in the column list. Using a Select statement instead of a list of value expressions lets you select a set of rows from one table and insert it into another table using a single Insert statement. The `SELECT` statement is evaluated before any values are inserted. This query cannot be made on the table into which values are inserted.

ITERATE statement

The `ITERATE` statement is used to break one iteration and continue directly to the next iteration of the specified `WHILE`, `REPEAT`, or `FOR` statements.

```
ITERATE <label>
```

`label`

is the label of the `WHILE`, `REPEAT`, or `FOR` statement in the `ITERATE` statement is specified.

Note: The ITERATE statement is illegal if used outside of the named block.

LEAVE statements

The `LEAVE` statement is used exit the specified iteration.

```
LEAVE <label>
```

`label`

is the label of the iterative statement that you want to exit.

MERGE Statements

The `MERGE` statement updates the records in a target table to match those in the source table by using update, insert and delete operations with a single statement.

```
MERGE INTO <target_table> USING <source_table> ON <expression> THEN <procedural_statement>
```

where:

`target_table`

is the table in which data is being inserted, updated, deleted.

`source_table`

is the name of the source table that the target table is being updated to match.

`expression`

the expression evaluated to determine whether to execute the procedural statement.

`procedural_statement`

is the procedural statement to be executed when the conditions in the expression are encountered.

LOOP statements

`LOOP` statements iterate the specified procedural statement until a `LEAVE` or other statement exits the loop.

```
[<label> :] LOOP <procedural_statement> END LOOP [<label>]
```

`label`

(optional) is the identifier for the `LOOP` statement that can be referenced by other statements. If specified, the label at the beginning of the statement must match the label at the end.

`procedural_statement`

is the procedural statement to be iterated until the loop is exited.

See also

LEAVE statements on page 259

OPEN statements

The `OPEN` statement opens a cursor and adds it to the list of result sets returned by the procedure.

```
OPEN <cursor_name>
```

`cursor_name`

is the name of a cursor you want to open. The cursor must have been declared by the `DECLARE CURSOR` statement earlier in the procedure.

See also

Cursor declarations on page 255

REPEAT statements

The `REPEAT` statement evaluates the condition in the expression after executing the specified statement. If the condition in the expression is met, the statement is executed again. This unlike the `WHILE` statement, which evaluates the condition before executing the statement.

```
[<label> :"] REPEAT <procedural_statement> UNTIL <expression> END REPEAT [<label>]
```

`label`

(optional) is the identifier for the `REPEAT` statement that can be referenced by other statements. If specified, the label at the beginning of the statement must match the label at the end.

`expression`

is the expression that is evaluated to determine whether the condition is met.

`procedural_statement`

is the procedural statement that is executed before evaluating the expression.

RESIGNAL statements

The `RESIGNAL` statement is used to return an exception from a handler.

```
RESIGNAL SQLSTATE <sql-state> [SET <message_text> = <expression>]
```

`sql_state`

specifies the five-character SQLSTATE code. For example, `HY000`.

`message_text`

(optional) specifies the message to text to be returned when the exception is thrown.

`expression`

(optional) is an expression that when met returns the message text.

Note: The message text is provided by the SQLSTATE unless the expression of the `SET` command is met.

SET statements

The SET statement allows you to assign values for variables. The following syntax is supported for the assignment statement:

```
SET (<variable_name> = <expression> | (<variable_name>[, ...]) = <select_statement>)
```

where:

`variable_name`

is the name of the variable to which you want to assign a value.

`expression`

are the variables, operators, literals, and method calls used to compute a value for the variable.

`select_statement`

(optional) is the syntax of a `SELECT` statement used to return the value of the variable.

For example, in the following statement, the expression is evaluated and the result is placed for the `example` variable..

```
CREATE FUNCTION squared(n INTEGER) RETURNS INTEGER
    BEGIN ATOMIC
    DECLARE example INTEGER;
    SET example = n * n;
    RETURN example;
END
```

In this statement, the `SELECT` statement is used to return zero or one rows. If no rows are returned, no assignment is made. However, if it returns one row, each column in the result is assigned to the corresponding named variable. If more than one row is returned, an exception is raised.

```
CREATE PROCEDURE get_eldest_child(IN id BIGINT, OUT name VARCHAR(32), OUT age
INTEGER)
    READS SQL DATA
    BEGIN ATOMIC
    SET (name, age) = (SELECT TOP 1 firstname, (CURRENT_DATE - dob) YEAR FROM
people ORDER BY dob DESC);
END
```

SIGNAL statements

The `SIGNAL` statement is used to return an exception. A handler could potentially detect and act on exceptions defined by these statements.

```
SIGNAL SQLSTATE <sql_state> [SET <message_text> = <expression>]
```

`sql_state`

specifies the five-character SQLSTATE code. For example, `HY000`.

`message_text`

(optional) specifies the message to text to be returned when the exception is thrown.

`expression`

(optional) is the expression that is evaluated to determine whether the message text is returned.

Note: Default message text is provided by the SQLSTATE unless the expression of the `SET` command is met; however, this message might not be useful to your user. If the text provided by SQLSTATE is insufficient, you should provide your own message using the `SET` command.

Scalar declarations

Each scalar declaration statement can declare one or more variables

```
DECLARE <scalar_name> [, ...] <data_type> [DEFAULT <value>]
```

`scalar_name`

specifies the name for the scalar variable. You can define multiple variables by separating the names with a comma.

`data_type`

specifies the data type, including the precision and scale, for to be declared for the variable. See "Data types syntax" for a complete list of supported data types.

`value`

(optional) specifies the default value assigned to the scalar variable.

See also

Data types syntax on page 248

Singleton SELECT statements

The Singleton SELECT statement is used to return a single row from a query operation.

```
SELECT <expression> (, ...)
 INTO <name> [, ...]
 FROM <select_statement>
```

where:

`expression`

is an expression that is evaluated to determine the row to return.

`name`

is the identifier of the object in which to update with the row returned .

`select_statement`

is a `SELECT` statement that is used to select the data containing the row to be returned.

Note: The `SELECT` should return 0 rows if not updating or one row to update. Returning more than one row returns an exception.

Table declarations

The following syntax allows you to declare temporary local tables that exist only within the scope of the block. This allows the procedure to accumulate data to be returned to the caller for interim calculations.

```
DECLARE TABLE <table_name> (<column_name> <data_type>(<p>, <s>)[, ...])
```

where:

`table_name`

Specifies the name of the temporary table to be declared.

`column_name`

Specifies the name of a column in the temporary table.

`data_type`

Specifies the data type, including the precision and scale, for the results of the function. See "Data types syntax" for a complete list of supported data types.

See also

Data types syntax on page 248

UPDATE statements

An Update statement changes the value of columns in the selected rows of a table. The following syntax is supported for `UPDATE` statements:

```
UPDATE <table_name> SET <column_name> = <expression> [, ...] [WHERE <conditions>]
```

where:

`table_name`

is the name of the table for which you want to update values.

column_name

is the name of a column, the value of which is to be changed. Multiple column values can be changed in a single statement.

expression

is the new value for the column. The expression can be a constant value or a subquery that returns a single value. Subqueries must be enclosed in parentheses.

WHILE statements

The WHILE statement executes the specified statement as long as the condition defined in the expression is met.

```
[<label> :] WHILE <expression> DO <procedural-statement> END WHILE [<label>]
```

label

(optional) is the identifier for the WHILE statement that can be referenced by other statements. If specified, the label at the beginning of the statement must match the label at the end.

expression

is the expression that is evaluated to determine whether the condition is met.

procedural_statement

is the procedural statement that is executed when a condition is met.

URL-encoded values

When specifying URL endpoints to be mapped to your REST model in either the Autonomous REST Composer or Model file, you must use valid URL-encoded values. URL-encoding is the process of converting unsafe or unsupported characters in your string to a set of safe characters in the ASCII format. Encoding your string is required to make your URL spec-compliant and, therefore, readable and safe for transmission across the Internet.

URL-encoding specification defines a set of reserved characters that act as delimiters in endpoints that have special meaning. When encoding your string, you must replace these characters with the encoded equivalent if they are not intended to be used as a delimiter. For example, if your URL contained the reserved characters space and !, you would need to replace each instance of the space character with %20 and the ! character with the value %21.

Non-encoded value:

```
http://example.com/new customers!/
```

Encoded value:

```
http://example.com/new%20customers%21/
```

The following table documents the reserved characters and their encoded equivalent. For more information on URL encoding, refer to https://en.wikipedia.org/wiki/Percent-encoding.

Table 40: Reserved characters

Reserved character	Encoded characters
space	%20
!	%21
"	%22
#	%23
$	%24
%	%25
&	%26
'	%27
(	%28
)	%29
*	%2A
+	%2B
,	%2C
/	%2F
:	%2D
;	%3A
=	%3B
?	%3D
@	%40
[	%5B
]	%5D

Example Model file

The following is an example Model file that can be modified for your environment.

```
{
    //An entry that defines how HTTP response status codes are processed by the driver.

    "#http":[ { "#code":200, "#action":"FAIL", "#operation":"SELECT",
                "#match":"\"status\":\"error"", "#message":"{message}", },
              { "#code":200, "#action":"OK" },
              { "#code":400, "#action":"ZERO_ROWS" },
              { "#code":401, "#action":"REAUTHENTICATE" },
              { "#code":404, "#action":"ZERO_ROWS" },
              { "#code":429, "#action":"RETRY_AFTER" },
              { "#code":503, "#action":"RETRY_AFTER" }]

    //An entry for a custom authentication request.
    "#authentication" : [
        "api-key={customAuthParams[1]}",
        {
            "credentials": {
                "username": "{user}",
                "password": "{password}",
                "company": "{customAuthParams[2]}"
            }
        },
        "POST http://{serverName}/bearertoken",
        "HEADER Authentication=Bearer {/access-token}"
    ]

    // A simple GET request without parameters to sample:
    "countries":"http://example.com/country",

    // A GET request with a parameter in the path:
    "states":"http://example.com/states/get/{countryCode:USA}/all",

    // A GET request with parameters as arguments

"timeseries":"https://www.example.com/times/query?interval=5min&symbol=USA&function=TIME_WEEKLY",


    // A GET request with custom HTTP headers
      "people":{
           "#path": "http://example.com/people",
           "#headers":{
                "Accept":"application/calendar+json",
                "X-Subway-Payment":"token",
                "X-Laundry-Service":"dryclean",
                "X-Favorite-Food":"pizza"
           }
      },

    // A POST with parameters in the body
    "countries2": {
          "#path": "http://example.com/country",
          "#post": {
                "start_date":"2018-08-31",
                "end_date":"2018-09-01",
                "departments":"[engineering,marketing,sales]",
                "tags":"[blue,green,red]"
         }
    },

    // A GET with paging configured
    "products": {
          "#path": "http://example.com/products",
```

```
            "#maximumPageSize":1000,
            "#firstRowNumber":1,
            "#pageSizeParameter":"maxResults",
            "#rowOffsetParameter":"startAt"
        },
    }
```

8

Supported SQL statements and extensions

The driver provides support for the SQL statements and the SQL extensions described in this section. SQL extensions are denoted by an (EXT) in the topic title.

For details, see the following topics:

- Alter Session (EXT)
- Delete
- Insert
- Select
- Update
- SQL expressions
- Subqueries

Alter Session (EXT)

Purpose

Changes various attributes of a local or remote session. A local session maintains the state of the overall connection. A remote session maintains the state that pertains to a particular remote data source connection.

Syntax

```
ALTER SESSION SET attribute_name=value
```

where:

attribute_name

specifies the name of the attribute to be changed. Attributes apply to either local or remote sessions.

value

specifies the value for that attribute.

The following table lists the local and remote session attributes, and provides descriptions of each.

Table 41: Alter Session Attributes

Attribute Name	Session Type	Description
Current_Schema	Local	Sets the current schema for the local session. The current schema is the schema used when an identifier in a SQL statement is unqualified. The string value must be the name of a schema visible in the local session. For example: `ALTER SESSION SET CURRENT_SCHEMA=AUTOREST`
Stmt_Call_Limit	Local	Sets the maximum number of Web service calls the driver can make in executing a statement. It sets the default Web service call limit used by any statement on the connection. Executing this command on a statement overrides the previously set StmtCallLimit for the connection. The value specified must be a positive integer or `0`. The value `0` means that no call limit exists. For example: `ALTER SESSION SET STMT_CALL_LIMIT=150`

Ws_Call_Count	Remote	Resets the Web service call count of a remote session to the value specified. The value must be `0` or a positive integer. WS_Call_Count represents the total number of Web service calls made to the remote data source instance for the current session. For example:
		`ALTER SESSION SET autorest.WS_CALL_COUNT=0`
		The current value of WS_Call_Count can be obtained by referring to the System_Remote_Sessions system table (see SYSTEM_REMOTE_SESSIONS Catalog Table for details). For example:
		`SELECT * from information_schema.system_remote_sessions WHERE session_id = cursessionid()`

Delete

Purpose

The Delete statement is used to delete rows from a table.

Syntax

```
DELETE FROM table_name [WHERE search_condition]
```

where:

table_name

specifies the name of the table from which you want to delete rows.

search_condition

is an expression that identifies which rows to delete from the table.

Notes

- The Where clause determines which rows are to be deleted. Without a Where clause, all rows of the table are deleted, but the table is left intact. See "Where Clause" for information about the syntax of Where clauses. Where clauses can contain subqueries.

Example A

This example shows a Delete statement on the emp table.

```
DELETE FROM emp WHERE emp_id = 'E10001'
```

Each Delete statement removes every record that meets the conditions in the Where clause. In this case, every record having the employee ID `E10001` is deleted. Because employee IDs are unique in the employee table, at most, one record is deleted.

Example B

This example shows using a subquery in a Delete clause.

```
DELETE FROM emp WHERE dept_id = (SELECT dept_id FROM dept WHERE dept_name =
'Marketing')
```

The records of all employees who belong to the department named Marketing are deleted.

Notes

- To enable Insert, Update, and Delete, write operations must be configured for the endpoint in the Model file. See "Write Operations" for details.

See also

Where clause on page 277
Write operations on page 213

Insert

Purpose

The Insert statement is used to add new rows to a local table. You can specify either of the following options:

- List of values to be inserted as a new row
- Select statement that copies data from another table to be inserted as a set of new rows

Syntax

```
INSERT INTO table_name [(column_name[,column_name]...)] {VALUES (expression
[,expression]...) | select_statement}
```

table_name

is the name of the table in which you want to insert rows.

column_name

is optional and specifies an existing column. Multiple column names (a column list) must be separated by commas. A column list provides the name and order of the columns, the values of which are specified in the Values clause. If you omit a *column_name* or a column list, the value expressions must provide values for all columns defined in the table and must be in the same order that the columns are defined for the table. Table columns that do not appear in the column list are populated with the default value, or with NULL if no default value is specified.

expression

is the list of expressions that provides the values for the columns of the new record. Typically, the expressions are constant values for the columns. Character string values must be enclosed in single quotation marks ('). See "Literals" for more information.

select_statement

is a query that returns values for each *column_name* value specified in the column list. Using a Select statement instead of a list of value expressions lets you select a set of rows from one table and insert it into another table using a single Insert statement. The Select statement is evaluated before any values are inserted. This query cannot be made on the table into which values are inserted. See "Select" for information about Select statements.

Notes

- To enable Insert, Update, and Delete, write operations must be configured for the endpoint in the Model file. See "Write Operations" for details.

See also

Literals on page 284
Select on page 272
Write operations on page 213

Select

Purpose

Use the Select statement to fetch results from one or more tables.

Syntax

```
SELECT select_clause from_clause
[where_clause]
[groupby_clause]
```

```
 [having_clause]
 [{UNION [ALL | DISTINCT] |
  {MINUS [DISTINCT] | EXCEPT [DISTINCT]} |
  INTERSECT [DISTINCT]} select_statement]
 [limit_clause]
```

where:

select_clause

specifies the columns from which results are to be returned by the query. See "Select" for a complete explanation.

from_clause

specifies one or more tables on which the other clauses in the query operate. See "From" for a complete explanation.

where_clause

is optional and restricts the results that are returned by the query. See "Where clause" for a complete explanation.

groupby_clause

is optional and allows query results to be aggregated in terms of groups. See "Group By clause" for a complete explanation.

having_clause

is optional and specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). See "Having clause" for a complete explanation.

`UNION`

is an optional operator that combines the results of the left and right Select statements into a single result. See "Union operator" for a complete explanation.

`INTERSECT`

is an optional operator that returns a single result by keeping any distinct values from the results of the left and right Select statements. See "Intersect operator" for a complete explanation.

`EXCEPT | MINUS`

are synonymous optional operators that returns a single result by taking the results of the left Select statement and removing the results of the right Select statement. See "Except and Minus operators" for a complete explanation.

orderby_clause

is optional and sorts the results that are returned by the query. See "Order By clause" for a complete explanation.

limit_clause

is optional and places an upper bound on the number of rows returned in the result. See "Limit clause" for a complete explanation.

Select clause

Purpose

Use the Select clause to specify with a list of column expressions that identify columns of values that you want to retrieve or an asterisk (*) to retrieve the value of all columns.

Syntax

```
SELECT [{LIMIT offsetnumber | TOP number}] [ALL | DISTINCT] {* | column_expression
[[AS] column_alias] [,column_expression [[AS] column_alias], ...]}
```

where:

`LIMIT` *`offset number`*

creates the result set for the Select statement first and then discards the first number of rows specified by *`offset`* and returns the number of remaining rows specified by *`number`*. To not discard any of the rows, specify `0` for *`offset`*, for example, `LIMIT 0` *`number`*. To discard the first *`offset`* number of rows and return all the remaining rows, specify `0` for *`number`*, for example, `LIMIT` *`offset`*`0`.

`TOP` *`number`*

is equivalent to `LIMIT 0`*`number`*.

`column_expression`

can be simply a column name (for example, `last_name`). More complex expressions may include mathematical operations or string manipulation (for example, `salary * 1.05`). See "SQL expressions" for details. *`column_expression`* can also include aggregate functions. See "Aggregate functions" for details.

`column_alias`

can be used to give the column a descriptive name. For example, to assign the alias department to the column dep:

```
SELECT dep AS department FROM emp
```

`DISTINCT`

eliminates duplicate rows from the result of a query. This operator can precede the first column expression. For example:

```
SELECT DISTINCT dep FROM emp
```

Notes

- Separate multiple column expressions with commas (for example, `SELECT last_name, first_name, hire_date`).
- Column names can be prefixed with the table name or table alias. For example, `SELECT emp.last_name or e.last_name`, where `e` is the alias for the table `emp`.
- NULL values are not treated as distinct from each other. The default behavior is that all result rows be returned, which can be made explicit with the keyword `ALL`.

See also

SQL expressions on page 283

Aggregate functions

Aggregate functions can also be a part of a Select clause. Aggregate functions return a single value from a set of rows. An aggregate can be used with a column name (for example, `AVG(salary)`) or in combination with a more complex column expression (for example, `AVG(salary * 1.07)`).

The following table lists supported aggregate functions.

Note: Doubly nested aggregates, such as `SUM(COUNT(col1))`, are currently not permitted by the driver.

Table 42: Aggregate Functions

Aggregate	Returns
AVG	The average of the values in a numeric column expression. For example, `AVG(salary)` returns the average of all salary column values.
COUNT	The number of values in any field expression. For example, `COUNT(name)` returns the number of name values. When using `COUNT` with a field name, `COUNT` returns the number of non-NULL column values. A special example is `COUNT(*)`, which returns the number of rows in the set, including rows with NULL values.
MAX	The maximum value in any column expression. For example, `MAX(salary)` returns the maximum salary column value.
MIN	The minimum value in any column expression. For example, `MIN(salary)` returns the minimum salary column value.
SUM	The total of the values in a numeric column expression. For example, `SUM(salary)` returns the sum of all salary column values.

Example

The following example uses the `COUNT`, `MAX`, and `AVG` aggregate functions:

```
SELECT
     COUNT(amount) AS numOpportunities,
     MAX(amount) AS maxAmount,
     AVG(amount) AS avgAmount
FROM opportunity o INNER JOIN user u
     ON o.ownerId = u.id
WHERE o.isClosed = 'false' AND
     u.name = 'MyName'
```

From clause

Purpose

The From clause indicates the tables to be used in the Select statement.

Syntax

```
FROM table_name [table_alias] [,...]
```

where:

table_name

is the name of a table or a subquery. Multiple tables define an implicit inner join among those tables. Multiple table names must be separated by a comma. For example:

```
SELECT * FROM emp, dep
```

Subqueries can be used instead of table names. Subqueries must be enclosed in parentheses. See "Subquery in a From clause" for an example.

table_alias

is a name used to refer to a table in the rest of the Select statement. When you specify an alias for a table, you can prefix all column names of that table with the table alias.

Example

The following example specifies two table aliases, `e` for `emp` and `d` for `dep`:

```
SELECT e.name, d.deptName
FROM emp e, dep d
WHERE e.deptId = d.id
```

table_alias is a name used to refer to a table in the rest of the Select statement. When you specify an alias for a table, you can prefix all column names of that table with the table alias. For example, given the table specification:

```
FROM emp E
```

you may refer to the last_name field as E.last_name. Table aliases must be used if the Select statement joins a table to itself. For example:

```
SELECT * FROM emp E, emp F WHERE E.mgr_id = F.emp_id
```

The equal sign (=) includes only matching rows in the results.

Join in a From clause

Purpose

You can use a Join as a way to associate multiple tables within a Select statement. Joins may be either explicit or implicit. For example, the following is the example from the previous section restated as an explicit inner join:

```
SELECT * FROM emp INNER JOIN dep ON id=empId
SELECT e.name, d.deptName
FROM emp e INNER JOIN dep d ON e.deptId = d.id;
```

whereas the following is the same statement as an implicit inner join:

```
SELECT * FROM emp, dep WHERE emp.deptID=dep.id
```

Note: The `ON` clause in a join expression must evaluate to a true or false value.

Syntax

```
FROM table_name {RIGHT OUTER | INNER | LEFT OUTER | CROSS | FULL OUTER} JOIN table.key
 ON search-condition
```

Example

In this example, two tables are joined using `LEFT OUTER JOIN`. `T1`, the first table named includes nonmatching rows.

```
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.key = T2.key
```

If you use a `CROSS JOIN`, no `ON` expression is allowed for the join.

Subquery in a From clause

Subqueries can be used in the From clause in place of table references (*`table_name`*).

Example

```
SELECT * FROM (SELECT * FROM emp WHERE sal > 10000) new_emp, dept WHERE
new_emp.deptno = dept.deptno
```

See also

Subqueries on page 291

Where clause

Purpose

Specifies the conditions that rows must meet to be retrieved.

Syntax

```
WHERE expr1 rel_operator expr2
```

where:

`expr1`

is either a column name, literal, or expression.

`expr2`

is either a column name, literal, expression, or subquery. Subqueries must be enclosed in parentheses.

`rel_operator`

is the relational operator that links the two expressions.

Example

The following Select statement retrieves the first and last names of employees that make at least $20,000.

```
SELECT last_name, first_name FROM emp WHERE salary >= 20000
```

See also

Subqueries on page 291
SQL expressions on page 283

Group By clause

Purpose

Specifies the names of one or more columns by which the returned values are grouped. This clause is used to return a set of aggregate values.

Syntax

```
GROUP BY column_expression [,...]
```

where:

column_expression

> is either a column name or a SQL expression. Multiple values must be separated by a comma. If *column_expression* is a column name, it must match one of the column names specified in the Select clause. Also, the Group By clause must include all non-aggregate columns specified in the Select list.

Example

The following example totals the salaries in each department:

```
SELECT dept_id, sum(salary) FROM emp GROUP BY dept_id
```

This statement returns one row for each distinct department ID. Each row contains the department ID and the sum of the salaries of the employees in the department.

See also

Subqueries on page 291
SQL expressions on page 283

Having clause

Purpose

Specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). This clause is valid only if you have already defined a Group By clause.

Syntax

```
HAVING expr1rel_operatorexpr2
```

where:

expr1 | expr2

> can be column names, constant values, or expressions. These expressions do not have to match a column expression in the Select clause. See "SQL expressions" for details regarding SQL expressions.

rel_operator

is the relational operator that links the two expressions.

Example

The following example returns only the departments that have salaries totaling more than $200,000:

```
SELECT dept_id, sum(salary) FROM emp GROUP BY dept_id HAVING sum(salary) > 200000
```

See also

Subqueries on page 291
SQL expressions on page 283

Union operator

Purpose

Combines the results of two Select statements into a single result. The single result is all the returned rows from both Select statements. By default, duplicate rows are not returned. To return duplicate rows, use the All keyword (`UNION ALL`).

Syntax

```
select_statement
UNION [ALL | DISTINCT] | {MINUS [DISTINCT] | EXCEPT [DISTINCT]} | INTERSECT
[DISTINCT]select_statement
```

Notes

- When using the Union operator, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
UNION
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
UNION
SELECT salary, last_name FROM raises
```

Intersect operator

Purpose

Intersect operator returns a single result set. The result set contains rows that are returned by both Select statements. Duplicates are returned unless the Distinct operator is added.

Syntax

```
select_statement
INTERSECT [DISTINCT]
select_statement
```

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

- When using the Intersect operator, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
INTERSECT [DISTINCT]
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
INTERSECT
SELECT salary, last_name FROM raises
```

Except and Minus operators

Purpose

Return the rows from the left Select statement that are not included in the result of the right Select statement.

Syntax

```
select_statement
{EXCEPT [DISTINCT] | MINUS [DISTINCT]}
select_statement
```

where:

`DISTINCT`

eliminates duplicate rows from the results.

Notes

- When using one of these operators, the Select lists for each Select statement must have the same number of column expressions with the same data types and must be specified in the same order.

Example A

The following example has the same number of column expressions, and each column expression, in order, has the same data type.

```
SELECT last_name, salary, hire_date FROM emp
EXCEPT
SELECT name, pay, birth_date FROM person
```

Example B

The following example is *not* valid because the data types of the column expressions are different (`salary FROM emp` has a different data type than `last_name FROM raises`). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
EXCEPT
SELECT salary, last_name FROM raises
```

Order By clause

Purpose

The Order By clause specifies how the rows are to be sorted.

Syntax

`ORDER BY` *`sort_expression`* `[DESC | ASC] [,...]`

where:

`sort_expression`

is either the name of a column, a column alias, a SQL expression, or the positioned number of the column or expression in the select list to use.

The default is to perform an ascending (`ASC`) sort.

Example

To sort by `last_name` and then by `first_name`, you could use either of the following Select statements:

`SELECT emp_id, last_name, first_name FROM emp`

`ORDER BY last_name, first_name`

or

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY 2,3
```

In the second example, `last_name` is the second item in the Select list, so `ORDER BY 2,3` sorts by `last_name` and then by `first_name`.

See also

SQL expressions on page 283

Limit clause

Purpose

Places an upper bound on the number of rows returned in the result.

Syntax

```
LIMIT number_of_rows [OFFSET offset_number]
```

where:

number_of_rows

specifies a maximum number of rows in the result. A negative number indicates no upper bound.

`OFFSET`

specifies how many rows to skip at the beginning of the result set. *offset_number* is the number of rows to skip.

Notes

- In a compound query, the Limit clause can appear only on the final Select statement. The limit is applied to the entire query, not to the individual Select statement to which it is attached.

Example

The following example returns a maximum of 20 rows.

```
SELECT last_name, first_name FROM emp WHERE salary > 20000 ORDER BY dept_idc LIMIT
20
```

Update

Purpose

An Update statement changes the value of columns in the selected rows of a table.

Syntax

```
UPDATE table_name SET column_name = expression
[, column_name = expression] [WHERE conditions]
```

table_name

is the name of the table for which you want to update values.

column_name

is the name of a column, the value of which is to be changed. Multiple column values can be changed in a single statement.

expression

is the new value for the column. The expression can be a constant value or a subquery that returns a single value. Subqueries must be enclosed in parentheses.

Example A

The following example changes every record that meets the conditions in the Where clause. In this case, the salary and exempt status are changed for all employees having the employee ID `E10001`. Because employee IDs are unique in the emp table, only one record is updated.

```
UPDATE emp SET salary=32000, exempt=1

WHERE emp_id = 'E10001'
```

Example B

The following example uses a subquery. In this example, the salary is changed to the average salary in the company for the employee having employee ID `E10001`.

```
UPDATE emp SET salary = (SELECT avg(salary) FROM emp)

WHERE emp_id = 'E10001'
```

Notes

- To enable Insert, Update, and Delete, write operations must be configured for the endpoint in the Model file. See "Write Operations" for details.
- A Where clause can be used to restrict which rows are updated.

See also

Subqueries on page 291
Where clause on page 277
Write operations on page 213

SQL expressions

An expression is a combination of one or more values, operators, and SQL functions that evaluate to a value. You can use expressions in the Where, and Having of Select statements; and in the Set clauses of Update statements.

Expressions enable you to use mathematical operations as well as character string manipulation operators to form complex queries.

The driver supports both unquoted and quoted identifiers. An unquoted identifier must start with an ASCII alpha character and can be followed by zero

Quoted identifiers must be enclosed in double quotation marks (""). A quoted identifier can contain any Unicode character including the space character. The driver recognizes the Unicode escape sequence \uxxxx as a Unicode character. You can specify a double quotation mark in a quoted identifier by escaping it with a double quotation mark.

The maximum length of both quoted and unquoted identifiers is 128 characters.

Valid expression elements are:

- Column names
- Literals
- Operators
- Functions

Column names

The most common expression is a simple column name. You can combine a column name with other expression elements.

Literals

Literals are fixed data values. For example, in the expression `PRICE * 1.05`, the value 1.05 is a constant. Literals are classified into types, including the following:

- Binary
- Character string
- Date
- Floating point
- Integer
- Numeric
- Time
- Timestamp

The following table describes the literal format for supported SQL data types.

Table 43: Literal Syntax Examples

SQL Type	Literal Syntax	Example
BIGINT	*n* where *n* is any valid integer value in the range of the INTEGER data type	`12 or -34 or 0`
BOOLEAN	Min Value: 0 Max Value: 1	`0` `1`
DATE	DATE'*date*'	`'2010-05-21'`

SQL Type	Literal Syntax	Example
DATETIME	TIMESTAMP'*ts*'	`'2010-05-21 18:33:05.025'`

DECIMAL	*n.f*
	where:
	n
	is the integral part
	f
	is the fractional part
		`0.25`
		`3.1415`
		`-7.48`

DOUBLE	*n.f*Ex where: *n* is the integral part *f* is the fractional part *x* is the exponent	`1.2E0` or `2.5E40` or `-3.45E2` or `5.67E-4`
INTEGER	*n* where *n* is a valid integer value in the range of the INTEGER data type	`12` or `-34` or `0`
LONGVARBINARY	'*hex_value*'	`'000482ff'`
LONGVARCHAR	'*value*'	`'This is a string literal'`
TIME	TIME'*time*'	`'2010-05-21 18:33:05.025'`
VARCHAR	'*value*'	`'This is a string literal'`

Character string literals

Text specifies a character string literal. A character string literal must be enclosed in single quotation marks. To represent one single quotation mark within a literal, you must enter two single quotation marks. When the data in the fields is returned to the client, trailing blanks are stripped.

A character string literal can have a maximum length of 32 KB, that is, (32*1024) bytes.

Example

```
'Hello'
'Jim''s friend is Joe'
```

Numeric literals

Unquoted numeric values are treated as numeric literals. If the unquoted numeric value contains a decimal point or exponent, it is treated as a real literal; otherwise, it is treated as an integer literal.

Example

```
+1894.1204
```

Binary literals

Binary literals are represented with single quotation marks. The valid characters in a binary literal are 0-9, a-f, and A-F.

Example

```
'00af123d'
```

Date/Time literals

Date and time literal values are enclosed in single quotion marks ('*value*').

- The format for a Date literal is DATE'*date*'.
- The format for a Time literal is TIME'*time*'.
- The format for a Timestamp literal is TIMESTAMP'*ts*'.

Integer literals

Integer literals are represented by a string of numbers that are not enclosed in quotation marks and do not contain decimal points.

Notes

- Integer constants must be whole numbers; they cannot contain decimals.
- Integer literals can start with sign characters (+/-).

Example

```
1994 or -2
```

Operators

This section describes the operators that can be used in SQL expressions.

Note: Numeric operators are restricted to numeric types. Numeric operators do not support non-numeric types.

Unary operator

A unary operator operates on only one operand.

operator operand

Binary operator

A binary operator operates on two operands.

operand1 operator operand2

If an operator is given a null operand, the result is always null. The only operator that does not follow this rule is concatenation (||).

Arithmetic operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of this operation is also a numeric value. The + and - operators are also supported in date/time fields to allow date arithmetic. The following table lists the supported arithmetic operators.

Table 44: Arithmetic Operators

Operator	Purpose	Example
+ -	Denotes a positive or negative expression. These are unary operators.	`SELECT * FROM emp WHERE comm = -1`
* /	Multiplies, divides. These are binary operators.	`UPDATE emp SET sal = sal + sal * 0.10`
+ -	Adds, subtracts. These are binary operators.	`SELECT sal + comm FROM emp WHERE empno > 100`

Concatenation operator

The concatenation operator manipulates character strings. The following table lists the only supported concatenation operator.

Table 45: Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings.	`SELECT 'Name is' || ename FROM emp`

The result of concatenating two character strings is the data type VARCHAR.

Comparison operators

Comparison operators compare one expression to another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN (if one of the operands is NULL). The driver considers the UNKNOWN result as FALSE.

The following table lists the supported comparison operators.

Table 46: Comparison Operators

Operator	Purpose	Example
=	Equality test.	`SELECT * FROM emp WHERE sal = 1500`
!=<>	Inequality test.	`SELECT * FROM emp WHERE sal != 1500`
><	"Greater than" and "less than" tests.	`SELECT * FROM emp WHERE sal > 1500 SELECT * FROM emp WHERE sal < 1500`
>=<=	"Greater than or equal to" and "less than or equal to" tests.	`SELECT * FROM emp WHERE sal >= 1500 SELECT * FROM emp WHERE sal <= 1500`
LIKE	`%` and _ wildcards can be used to search for a pattern in a column. The percent sign denotes zero, one, or multiple characters, while the underscore denotes a single character. The right-hand side of a LIKE expression must evaluate to a string or binary.	`SELECT * FROM emp WHERE ENAME LIKE 'J%'`

ESCAPE clause in LIKE operator
LIKE 'pattern string' ESCAPE 'c'
	The Escape clause is supported in the LIKE predicate to indicate the escape character. Escape characters are used in the pattern string to indicate that any wildcard character that is after the escape character in the pattern string should be treated as a regular character.
	The default escape character is backslash (\).
		`SELECT * FROM emp WHERE ENAME LIKE 'J%_%' ESCAPE '\'`
		This matches all records with names that start with letter 'J' and have the '_' character in them.
		`SELECT * FROM emp WHERE ENAME LIKE 'JOE_JOHN' ESCAPE '\'`
		This matches only records with name 'JOE_JOHN'.

[NOT] IN	"Equal to any member of" test.	`SELECT * FROM emp WHERE job IN ('CLERK','ANALYST') SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30)`
[NOT] BETWEEN x AND y	"Greater than or equal to x" and "less than or equal to y."	`SELECT * FROM emp WHERE sal BETWEEN 2000 AND 3000`

Operator	Purpose	Example
EXISTS	Tests for existence of rows in a subquery.	`SELECT empno, ename, deptno FROM emp e WHERE EXISTS (SELECT deptno FROM dept WHERE e.deptno = dept.deptno)`
IS [NOT] NULL	Tests whether the value of the column or expression is NULL.	`SELECT * FROM emp WHERE ename IS NOT NULL SELECT * FROM emp WHERE ename IS NULL`

Logical operators

A logical operator combines the results of two component conditions to produce a single result or to invert the result of a single condition. The following table lists the supported logical operators.

Table 47: Logical Operators

Operator	Purpose	Example
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	`SELECT * FROM emp WHERE NOT (job IS NULL)` `SELECT * FROM emp WHERE NOT (sal BETWEEN 1000 AND 2000)`
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE; otherwise, returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' AND deptno = 10`
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE; otherwise, returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' OR deptno = 10`

Example

In the Where clause of the following Select statement, the `AND` logical operator is used to ensure that managers earning more than $1000 a month are returned in the result:

```
SELECT * FROM emp WHERE jobtitle = manager AND sal > 1000
```

Operator precedence

As expressions become more complex, the order in which the expressions are evaluated becomes important. The following table shows the order in which the operators are evaluated. The operators in the first line are evaluated first, then those in the second line, and so on. Operators in the same line are evaluated left to right in the expression.You can change the order of precedence by using parentheses. Enclosing expressions in parentheses forces them to be evaluated together.

Table 48: Operator Precedence

Precedence	Operator
1	+ (Positive), - (Negative)
2	*(Multiply), / (Division)
3	+ (Add), - (Subtract)
4	|| (Concatenate)
5	=, >, <, >=, <=, <>, != (Comparison operators)
6	NOT, IN, LIKE
7	AND
8	OR

Example A

The query in the following example returns employee records for which the department number is 1 or 2 and the salary is greater than $1000:

```
SELECT * FROM emp WHERE (deptno = 1 OR deptno = 2) AND sal > 1000
```

Because parenthetical expressions are forced to be evaluated first, the OR operation takes precedence over AND.

Example B

In the following example, the query returns records for all the employees in department 1, but only employees whose salary is greater than $1000 in department 2.

```
SELECT * FROM emp WHERE deptno = 1 OR deptno = 2 AND sal > 1000
```

The AND operator takes precedence over OR, so that the search condition in the example is equivalent to the expression `deptno = 1 OR (deptno = 2 AND sal > 1000).`

Functions

The driver supports a number of functions that you can use in expressions.

Refer to "Scalar functions" in the *Progress DataDirect for ODBC Drivers Reference* for more information.

Conditions

A condition specifies a combination of one or more expressions and logical operators that evaluates to either TRUE, FALSE, or UNKNOWN. You can use a condition in the Where clause of the Delete, Select, and Update statements; and in the Having clauses of Select statements. The following describes supported conditions.

Table 49: Conditions

Condition	Description
Simple comparison	Specifies a comparison with expressions or subquery results. `= , !=, <>, < , >, <=, <=`
Group comparison	Specifies a comparison with any or all members in a list or subquery. `[= , !=, <>, < , >, <=, <=] [ANY, ALL, SOME]`
Membership	Tests for membership in a list or subquery. `[NOT] IN`
Range	Tests for inclusion in a range. `[NOT] BETWEEN`
NULL	Tests for nulls. `IS NULL, IS NOT NULL`
EXISTS	Tests for existence of rows in a subquery. `[NOT] EXISTS`
LIKE	Specifies a test involving pattern matching. `[NOT] LIKE`
Compound	Specifies a combination of other conditions. `CONDITION [AND/OR] CONDITION`

Subqueries

A query is an operation that retrieves data from one or more tables or views. In this reference, a top-level query is called a Select statement, and a query nested within a Select statement is called a subquery.

A subquery is a query expression that appears in the body of another expression such as a Select, an Update, or a Delete statement. In the following example, the second Select statement is a subquery:

```
SELECT * FROM emp WHERE deptno IN (SELECT deptno FROM dept)
```

IN predicate

Purpose

The In predicate specifies a set of values against which to compare a result set. If the values are being compared against a subquery, only a single column result set is returned.

Syntax

```
value [NOT] IN (value1, value2,...)
```

OR

```
value [NOT] IN (subquery)
```

Example

```
SELECT * FROM emp WHERE deptno IN

(SELECT deptno FROM dept WHERE dname <> 'Sales')
```

EXISTS predicate

Purpose

The Exists predicate is true only if the cardinality of the subquery is greater than 0; otherwise, it is false.

Syntax

```
EXISTS (subquery)
```

Example

```
SELECT empno, ename, deptno FROM emp e WHERE EXISTS
(SELECT deptno FROM dept WHERE e.deptno = dept.deptno)
```

UNIQUE predicate

Purpose

The Unique predicate is used to determine whether duplicate rows exist in a virtual table (one returned from a subquery).

Syntax

```
UNIQUE (subquery)
```

Example

```
SELECT * FROM dept d WHERE UNIQUE
(SELECT deptno FROM emp e WHERE e.deptno = d.deptno)
```

Correlated subqueries

Purpose

A correlated subquery is a subquery that references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a Select, Update, or Delete statement.

A correlated subquery answers a multiple-part question in which the answer depends on the value in each row processed by the parent statement. For example, you can use a correlated subquery to determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

Syntax

```
SELECT select_list
   FROM table1 t_alias1
   WHERE expr rel_operator
   (SELECT column_list
       FROM table2 t_alias2
       WHERE t_alias1.columnrel_operatort_alias2.column)
UPDATE table1 t_alias1
   SET column =
   (SELECT expr
       FROM table2 t_alias2
        WHERE t_alias1.column = t_alias2.column)
DELETE FROM table1 t_alias1
   WHERE column rel_operator
   (SELECT expr
       FROM table2 t_alias2
       WHERE t_alias1.column = t_alias2.column)
```

Notes

- Correlated column names in correlated subqueries must be explicitly qualified with the table name of the parent.

Example A

The following statement returns data about employees whose salaries exceed their department average. This statement assigns an alias to `emp`, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT deptno, ename, sal FROM emp x WHERE sal >
  (SELECT AVG(sal) FROM emp WHERE x.deptno = deptno)
  ORDER BY deptno
```

Example B

This is an example of a correlated subquery that returns row values:

```
SELECT * FROM dept "outer" WHERE 'manager' IN
  (SELECT managername FROM emp
  WHERE "outer".deptno = emp.deptno)
```

Example C

This is an example of finding the department number (deptno) with multiple employees:

```
SELECT * FROM dept main WHERE 1 <
  (SELECT COUNT(*) FROM emp WHERE deptno = main.deptno)
```

Example D

This is an example of correlating a table with itself:

```
SELECT deptno, ename, sal FROM emp x WHERE sal >
  (SELECT AVG(sal) FROM emp WHERE x.deptno = deptno)
```

9

Pre-defined stored procedures

In the prebuilt Model files, files for the data stores described in this section include pre-defined stored procedures in addition to the required requests and pagination parameters. These functions can be called by the application to access additional functionality that would not otherwise be available to the driver, such as the ability to fetch, insert, update, delete documents stored in document stores.

See the following sections for a list of supported functions by data source:

- Amazon S3
- Box
- Dropbox
- Google Drive
- Microsoft Azure Data Lake Storage

For details, see the following topics:

- Amazon S3 stored procedures
- Box stored procedures
- Dropbox stored procedures
- Google Drive stored procedures
- Microsoft Azure Data Lake stored procedures

Amazon S3 stored procedures

The following stored procedures are supported for AWS S3.

Table 50: Required fields

Procedure Name	Description
copyObject	Copies an object to another specified bucket.
downloadObject	Downloads the contents of an object.
uploadObject	Uploads a new object and its contents.

copyObject

Purpose

Copies an object to another specified bucket. Note that both buckets must exist in the same region; therefore, copying across regions is not supported.

Table 51: Required fields

Field Name	Data Type	Parameter Type	Description
`destinationBucket`	VARCHAR(255)	IN	Specifies the name of the bucket to which the object is copied.
`destinationFile`	VARCHAR(255)	IN	Specifies the name of the copy file.
`sourceFile`	VARCHAR(255)	IN	Specifies the name of the file to be copied.
`optionalHeaders`	VARCHAR(255)	IN	Specifies the a comma separated list of optional headers accepted by the endpoint, as detailed in the AWS documentation. For example, `x-amz-copy-source-if-match=true`.

Table 52: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

downloadObject

Purpose

Downloads the contents of an object.

Table 53: Required fields

Field Name	Data Type	Parameter Type	Description
`bucket`	VARCHAR(255)	IN	Specifies the name of the bucket containing the file to be downloaded.
`fileName`	VARCHAR(255)	IN	Specifies the name of the file to be downloaded.

Table 54: Response

Response	Data Type	Parameter Type	Description
`content`	VARBINARY(16777215)	OUT	The content body of the file.

uploadObject

Purpose

Uploads a new object and its contents.

Table 55: Required fields

Field Name	Data Type	Parameter Type	Description
`bucket`	VARCHAR(255)	IN	Specifies the name of the bucket where the object should be uploaded.
`fileName`	VARCHAR(255)	IN	Specifies the name of the object to be uploaded.
`content`	VARCHAR(255)	IN	Specifies the content of the object to be uploaded.

Table 56: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

Box stored procedures

The following stored procedures are supported for Box cloud storage.

Table 57: Required fields

Procedure Name	Description
copyFile	Copies a file to the specified location.
copyFolder	Copies a folder to the specified location.
downloadObject	Downloads the specified file.
uploadObject	Uploads a new file to Box storage if it does not already exist.

copyFile

Purpose

Copies a file to a specified location.

Table 58: Required fields

Field Name	Data Type	Parameter Type	Description
`fileName`	VARCHAR(64)	IN	Specifies the name of the copy file.
`fileId`	VARCHAR(64)	IN	Specifies the ID of the file to be copied.
`parentFolderId`	VARCHAR(64)	IN	Specifies the ID of the folder in which the file will be copied.

Table 59: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

copyFolder

Purpose

Copies a folder to a specified location.

Table 60: Required fields

Field Name	Data Type	Parameter Type	Description
`folderName`	VARCHAR(64)	IN	Specifies the name of the copy folder.
`folderId`	VARCHAR(64)	IN	Specifies the ID of the folder to be copied.
`parentFolderId`	VARCHAR(64)	IN	Specifies the name of the folder to which the folder is copied.

Table 61: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

downloadObject

Purpose

Downloads the specified file.

Table 62: Required fields

Field Name	Data Type	Parameter Type	Description
`fileId`	VARCHAR(64)	IN	Specifies the ID of the file to be downloaded.

Table 63: Response

Response	Data Type	Parameter Type	Description
`entity`	VARBINARY(16777215)	OUT	The content body of the file.

uploadObject

Purpose

Uploads a new file to Box storage if it does not already exist.

Table 64: Required fields

Field Name	Data Type	Parameter Type	Description
`fileName`	VARCHAR(64)	IN	Specifies the name of the object to be uploaded.

Field Name	Data Type	Parameter Type	Description
`entity`	VARBINARY(16777215)	IN	Specifies the contents of the file to be uploaded.
`parentFolderID`	VARCHAR(64)	IN	Specifies the content of the object to be uploaded.

Table 65: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

Dropbox stored procedures

The following stored procedures are supported for Dropbox file hosting service.

Table 66: Required fields

Procedure Name	Description
deleteFile	Deletes the file from the specified path.
downloadFile	Downloads the contents of a file.
uploadFile	Uploads a new file and its contents.

deleteFile

Purpose

Deletes a file from the specified path.

Table 67: Required fields

Field Name	Data Type	Parameter Type	Description
`path`	OTHER	IN	Specifies the name and path of the file to be deleted. For example, `{"path":"/Documents/myFile.txt"}`.

Table 68: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

downloadFile

Purpose

Downloads the contents of a file.

Table 69: Required fields

Field Name	Data Type	Parameter Type	Description
`path`	VARCHAR(255)	IN	Specifies the name and path of the file to be downloaded. For example, `/path/to/myFile.txt`.

Table 70: Response

Response	Data Type	Parameter Type	Description
`content`	VARBINARY(16777215)	OUT	The content body of the file.

uploadFile

Purpose

Uploads a new file and its contents.

Table 71: Required fields

Field Name	Data Type	Parameter Type	Description
`path`	VARCHAR(255)	IN	Specifies the destination name and path of the uploaded file.
`autorename`	VARCHAR(255)	IN	Specifies whether you want Dropbox to attempt to rename the uploaded file if there is a name conflict. If set to `True`, Dropbox attempts to rename the file to be uploaded if a potential naming conflict is detected. If set to `False`, Dropbox does not attempt to rename the file if a potential naming conflict is detected, and the upload fails.
`content`	OTHER	IN	Specifies the content of the object to be uploaded.

Table 72: Response

Response	Data Type	Parameter Type	Description
status	INTEGER	OUT	The response code for the request.

Google Drive stored procedures

The following stored procedures are supported for Google Drive.

Table 73: Required fields

Procedure Name	Description
downloadObject	Downloads the contents of a file.
uploadObject	Uploads new contents to an existing file.

downloadObject

Purpose

Downloads the contents of a file.

Table 74: Required fields

Field Name	Data Type	Parameter Type	Description
fileID	VARCHAR(255)	IN	Specifies the ID of the file from where you want to download the content.

Table 75: Response

Response	Data Type	Parameter Type	Description
entity	VARBINARY(16777215)	OUT	The content body in binary format.

uploadObject

Purpose

Uploads new contents to an existing file.

Table 76: Required fields

Field Name	Data Type	Parameter Type	Description
`fileId`	VARCHAR(256)	IN	Specifies the ID of the file in which you want to load the content
`fileName`	VARCHAR(256)	IN	Specifies the name of the file in which you want to load content
`content`	OTHER	IN	Specifies the content of the file you are uploading

Table 77: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request

Microsoft Azure Data Lake stored procedures

The following stored procedures are supported for Microsoft Azure Data Lake Storage Gen2.

Table 78: Required fields

Procedure Name	Description
appendFile	Appends content that is being prepared to be flushed to the specified file.
copyFile	Copies a file within a file system to another specified location.
createFile	Creates an empty file in the specified location.
createFolder	Creates an empty folder in the specified location.
downloadFile	Downloads the specified file and is returned in the response for the user.
flushFile	Flushes the content of the specified file.
uploadFile	Uploads a file to the specified location. If the file already exists, it is overwritten.
writeFileContents	Writes file contents to the specified file.

appendFile

Purpose

Appends content that is being prepared to be flushed to the specified file.

Table 79: Required fields

Field Name	Data Type	Parameter Type	Description
`accountName`	VARCHAR(64)	IN	Specifies the name of the storage account
`fileSystem`	VARCHAR(64)	IN	Specifies the file system to which the file will be uploaded
`content`	OTHER	IN	Specifies the content of the file
`filePath`	VARCHAR(128)	IN	Specifies the name and path of the file to which content is appended. For example, `path/to/file`.

Table 80: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

copyFile

Purpose

Copies a file within a file system to another specified location.

Table 81: Required fields

Field Name	Data Type	Parameter Type	Description
`accountName`	VARCHAR(64)	IN	Specifies the name of the storage account
`fileSystem`	VARCHAR(64)	IN	Specifies the file system to which the file will be copied
`sourceFilePath`	VARCHAR(128)	IN	Specifies the name and path of the file name to be copied
`destinationFilePath`	VARCHAR(128)	IN	Specifies the name and path of the copy of the file. For example, `path/to/file`.

Table 82: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

createFile

Purpose

Creates an empty file in the specified location.

Table 83: Required fields

Field Name	Data Type	Parameter Type	Description
`accountName`	VARCHAR(64)	IN	Specifies the name of the storage account
`fileSystem`	VARCHAR(64)	IN	Specifies the file system in which the file will be created
`filePath`	VARCHAR(128)	IN	Specifies the name and path of the folder to be created. For example, `path/to/folder`.

Table 84: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

createFolder

Purpose

Creates an empty folder in the specified location.

Table 85: Required fields

Field Name	Data Type	Parameter Type	Description
`accountName`	VARCHAR(64)	IN	Specifies the name of the storage account
`fileSystem`	VARCHAR(64)	IN	Specifies the file system in which the file will be created
`directoryPath`	VARCHAR(128)	IN	Specifies the name and path of the folder to be created. For example, `path/to/folder`.

Table 86: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

downloadFile

Purpose

Downloads the specified file and is returned in the response for the user.

Table 87: Required fields

Field Name	Data Type	Parameter Type	Description
`accountName`	VARCHAR(64)	IN	Specifies the name of the storage account
`fileSystem`	VARCHAR(64)	IN	Specifies the file system in which the file will be created
`filePath`	VARCHAR(128)	IN	Specifies the name and path of the file. For example, `path/to/file`.

Table 88: Response

Response	Data Type	Parameter Type	Description
`entity`	VARBINARY(16777215)	OUT	Returns the response, as VARBINARY, from the download request.

flushFile

Purpose

Appends content that is being prepared to be flushed to the specified file.

Table 89: Required fields

Field Name	Data Type	Parameter Type	Description
`contentLength`	INTEGER	IN	Specifies the length of the file
`accountName`	VARCHAR(64)	IN	Specifies the name of the storage account
`fileSystem`	VARCHAR(64)	IN	Specifies the file system to which the file will be uploaded
`filePath`	VARCHAR(128)	IN	Specifies the name and path of the file to which content is appended. For example, `path/to/file`.

Table 90: Response

Response	Data Type	Parameter Type	Description
status	INTEGER	OUT	The response code for the request.

uploadFile

Purpose

Uploads a file to the specified location. If the file already exists, it is overwritten.

Table 91: Required fields

Field Name	Data Type	Parameter Type	Description
contentLength	INTEGER	IN	Specifies, in bytes, the content length of the file to be uploaded
content	OTHER	IN	Specifies the content of the file
accountName	VARCHAR(64)	IN	Specifies the name of the storage account
fileSystem	VARCHAR(64)	IN	Specifies the file system in which the file will be created
filePath	VARCHAR(128)	IN	Specifies the name and path of the file. For example, `path/to/folder`.

Table 92: Returns

Field Name	Data Type	Parameter Type	Description
status	INTEGER	OUT	The response code for the request.

writeFileContents

Purpose

Writes file contents to the specified file.

Table 93: Required fields

Field Name	Data Type	Parameter Type	Description
contentLength	INTEGER	IN	Specifies, in bytes, the content length of the file to be uploaded
content	OTHER	IN	Specifies the content of the file

Field Name	Data Type	Parameter Type	Description
`accountName`	VARCHAR(64)	IN	Specifies the name of the storage account
`fileSystem`	VARCHAR(64)	IN	Specifies the file system in which the file will be created
`filePath`	VARCHAR(128)	IN	Specifies the name and path of the file to which content is to be written. For example, `path/to/file`.

Table 94: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.