Progress DataDirect

Progress DataDirect Autonomous REST Connector for JDBC

User's Guide

Release 6.0.1

Copyright

Table of Contents

1

Welcome to the Progress DataDirect Autonomous REST Connector for JDBC

The Progress DataDirect Autonomous REST Connector for JDBC is a driver that supports SQL read and update access to JSON, XML, and CSV 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 can 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 JDBC 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 the driver is 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 on page 18
- Getting started creating a Model file on page 21
- Installing and setting up the driver on page 25

The documentation for the driver also includes the *Progress DataDirect for JDBC Drivers Reference*. The reference provides general reference information for all DataDirect drivers for JDBC, including content on troubleshooting, supported SQL escapes, and DataDirect tools.

For the complete documentation set, visit the Progress DataDirect Connectors Documentation Hub: https://docs.progress.com/category/datadirect-autonomous-rest-connector.

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

For details, see the following topics:

What's new in this release?

Support and certification

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

- Release Notes: https://www.progress.com/datadirect-connectors/whats-new#jdbc
- DataDirect Product Compatibility Guide: https://docs.progress.com/bundle/datadirect-product-compatibility/resource/datadirect-product-compatibility.pdf

Changes Since 6.0.1 Release

- **Driver Enhancements**
  - The driver has been enhanced to support sending paging parameters in the body of a POST request, rather than as query parameters, for Select statements that use POST instead of GET requests. This behavior may be enabled with the PostBodyPaging parameter. See Paging on page 197 for details.
  - The driver has been enhanced to support the upload and download of large files to and from Amazon S3 using the multipart upload/download feature. You can configure this feature using the connection

property FileTransferPartSize and stored procedures: getObjectSize, uploadFile, and downloadFile. See FileTransferPartSize on page 141 and Amazon S3 stored procedures on page 278 for more details.

- The driver has been enhanced to support branding or white labeling the Autonomous REST Composer. Branding allows you to apply logos, names, and styles across error messages, web interfaces, and documentation. For more information, refer to Branding the Autonomous REST Composer in the *Progress DataDirect for JDBC Drivers Distribution Guide*.
- When deploying the Autonomous REST Connector in a SaaS application, you may enable diagnostic recording and download log files directly from the Composer interface. For more information, refer to Diagnostic recording in the Autonomous REST Composer in the *Progress DataDirect for JDBC Drivers Reference*.
- The QualifyNormalizedNames property has been added to the driver. This property 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 QualifyNormalizedNames on page 158.
- 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 `POST` setting for the ClientCredentialsMode property. See ClientCredentialsMode on page 130 and OAuth 2.0 authentication on page 84 for more details.
- The driver has been enhanced to comply with FIPS standards for data encryption. As part of this enhancement, the driver was tested with FIPS 140-3 enabled using a Red Hat OpenJDK 21 on a Red Hat Universal Base Image 9 instance. See FIPS (Federal Information Processing Standard) on page 105 for details.
- 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 39 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 JSONRoot property. See Sampling REST endpoints on page 63 and JSONRoot on page 146 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 SamplingFailureTolerence property, you can specify the number of endpoints for which sampling can fail before the driver fails the connection. See SamplingFailureTolerance on page 165 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 69 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 ClientCredentialsMode property. See ClientCredentialsMode on page 130 and OAuth 2.0 authentication on page 84 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 DefaultQueryOptions property. See DefaultQueryOptions on page 137 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 197 for more details.
- The Autonomous REST Composer has been enhanced with a new SQL Editor tool. The new SQL Editor allows you to execute test queries with an improved interface, which includes a view of your schemas, table details, and results. See Testing and querying your Model on page 68 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 AuthenticationMethod property, 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 AuthenticationMethod property will continue to be supported. See OAuth 2.0 authentication on page 84 and AuthenticationMethod on page 125 for details.
- The driver has been enhanced to support the PKCE grant type for OAuth 2.0 authentication. See PKCE grant on page 95 for more details.
- The driver has been enhanced to the support the JWT (JSON Web Token) bearer grant type for OAuth 2.0 authentication. You can configure the JWT bearer grant authentication using the new ClaimsIssuer, ClaimsSubject, JWTCertAlias, JWTCertPassword, and JWTCertStore properties. See JWT bearer grant on page 92 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 197 for details.
- The driver has been enhanced to support fetching access and refresh tokens at connection when OAuth 2.0 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 90 and EnableLoginPrompt on page 138 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 208 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 277 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 226 for more supported functionality and syntax.
- Update operation functionality has been enhanced to support APIs that require using `PATCH`, `PUT`, or `POST` methods that send only changed fields in the body. You can configure update operations using the `#update` and `#sendOnlyUpdated` parameters in the Model file. See Write operations on page 195 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 219 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 220 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 195 and Supported SQL statements and extensions on page 251 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 65 for more information.
- The driver has been updated with the new JSONRoot connection property, 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 JSONRoot on page 146 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 59 for more information.
- 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 188.
- The driver now includes a library of Progress developed Model files to connect to publicly accessible REST services. The prebuilt Model files fully 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 on page 18 for details.
- The Configuration Manager has been enhanced with role-specific work flows:
  - The developer view of the Configuration Manager, known as the Autonomous REST Composer, provides access to the new prebuilt Model library, REST management tools, 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 by double-clicking on the `autorest.jar` file or launching it from a command prompt.

- 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 31 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 AuthHeader property and the ID value with the SecurityToken property. This functionality can be used for passing the ID string for tenant ID authentication. See OAuth 2.0 authentication on page 84 for details.
- The driver has been enhanced to support AWS (Amazon Web Services) credentials authentication. When AWS credentials authentication is enabled (`AuthenticationMethod=AWS`), you can configure AWS credentials using the new AccessKey, Region, and SecretKey properties. See AWS credentials authentication on page 80 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 206 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 parameters on page 211 for details.

- Changed behavior
    - The connection property `SpyAttributes` has been updated to exclude the attribute `load=classname`, which was previously used to load the driver specified by the given class name. See SpyAttributes on page 168 for details.
    - The behavior of the ClientCredentialsMode property has been updated:
        - The `All` setting has been replaced with the `Default` setting.
        - If set to `Default`, the client credentials are sent as a basic authentication header.
        - The default setting for the ClientCredentialsMode has changed from `All` to `Default`.

      See ClientCredentialsMode on page 130 for more supported functionality and syntax.
    - 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 6.0.1 Release

- **Driver Enhancements**
    - On Windows, the driver now includes the DataDirect JDBC Driver Configuration Manager for quick configuration and testing of your driver. This Configuration Manager allows you to:

    - Generate and edit connection URLs
    - Test connect connection URLs
    - Execute SQL commands for testing
    - Fetch OAuth tokens and configure OAuth
    - Access connection property descriptions and the full product documentation
- The Data Direct JDBC Configuration Manger has been enhanced to support the generation of Model files. This provides you with a method to quickly generate and edit a Model file. See Generating a Model file with the Autonomous REST Composer on page 59 for details.
- The driver has been enhanced to support bearer token and digest authentication. See Bearer token authentication on page 81 and Digest authentication on page 82 for more details.
- Interactive SQL is now installed with the product. Interactive SQL is a command-line interface that supports connecting your driver to a data source, executing SQL statements and retrieving results in a terminal. This tool provides a method to quickly test your drivers in an environment that does not support GUIs. See Interactive SQL on page 53 for details.
- The new HealthURI connection property provides a method to allow you 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 HealthURI on page 142 for details.
- The new TransactionMode connection property allows you to determine how the driver handles manual transactions. See TransactionMode on page 174 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 197 for details.
- The driver has been enhanced to include timestamp in the Spy and JDBC packet logs by default. See SpyAttributes on page 168 for details.
- The driver has been enhanced to allow you to define custom authentication requests, including the new CustomAuthParams connection property. If your service does not support one of the standard authentication methods supported by the driver, you can modify the Model file to define a custom authentication flow. See Custom authentication on page 100 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 183 for details.
- The driver has been enhanced to support OAuth 2.0 authentication. See OAuth 2.0 authentication on page 84 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 210 for details.

- **Changed behavior**:
    - The CreateMap and SchemaMap properties are no longer supported. The driver now generates the schema map for each session and stores it internal memory.
    - The LoginTimeout property is no longer supported. The LoginTimeout property was used to specify the amount of time that the driver waited for a connection to be established before timing out the connection request.

Highlights of 6.0.0 Release

- The driver supports SQL read-only access to REST API endpoints returning JSON payloads. See Supported SQL statements and extensions on page 251 for details.
- The driver supports JDBC core functions. For details, refer to "JDBC support" in the *Progress DataDirect for JDBC Drivers Reference*.
- The driver supports standard JSON data types and additional data types through data type inference. See Data types on page 40 and getTypeInfo() on page 41 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 58 for details.
- The driver heuristically maps data types, eliminating the need to define native data types in most scenarios. See Data types on page 40 for more information about data type mapping.
- The driver supports basic, HTTP-header based, URL-Parameter based and no authentication. See Authentication on page 78 for details.
- The driver supports the handling of large result sets with configurable paging and the FetchSize on page 140 and WSFetchSize on page 177 connection properties. See Configuring the relational map on page 58 for more information on configuring paging.

Requirements

The driver is compatible with JDBC 2.0, 3.0, and 4.0.

The driver requires a Java Virtual Machine (JVM) that is Java SE 8 or higher, including Oracle JDK, OpenJDK, and IBM SDK (Java) distributions.

Getting started using prebuilt Model files

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 selecting a Model, you only need to provide your authentication credentials to begin accessing data.

To begin accessing data with the driver:

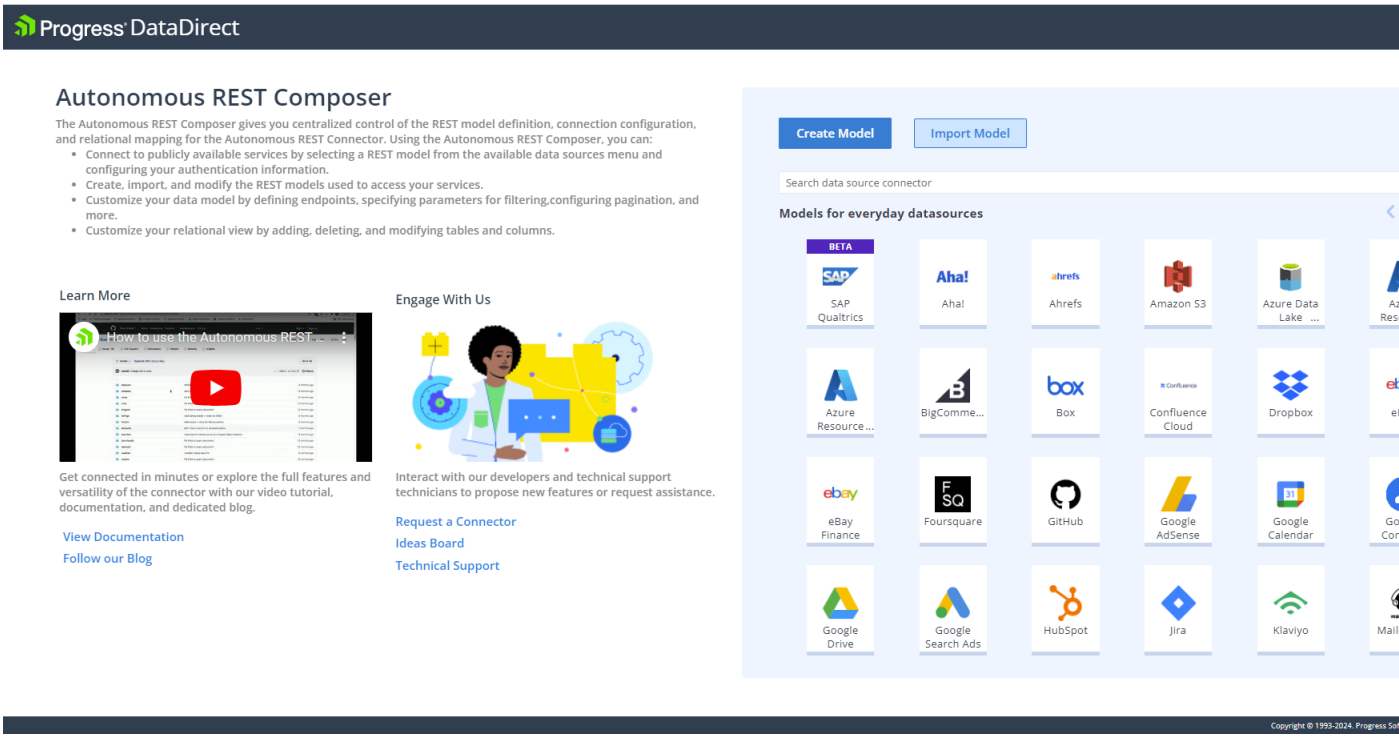
1. Install the driver:
   a) After downloading the product, unzip the installer files to a temporary directory.
   b) From the installer directory, run the appropriate installer file to start the installer.
      - **Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.exe`
      - **Non-Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.jar`
   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 JDBC Drivers Installation Guide*.
2. Open the Autonomous REST Composer by using one of the following methods:

- Select the **Autonomous REST Composer (JDBC)** 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 --design
  ```

  By default, the `autorest.jar` file is stored in the following directory: `C:\Program Files\Progress\DataDirect\JDBC\lib\60`.

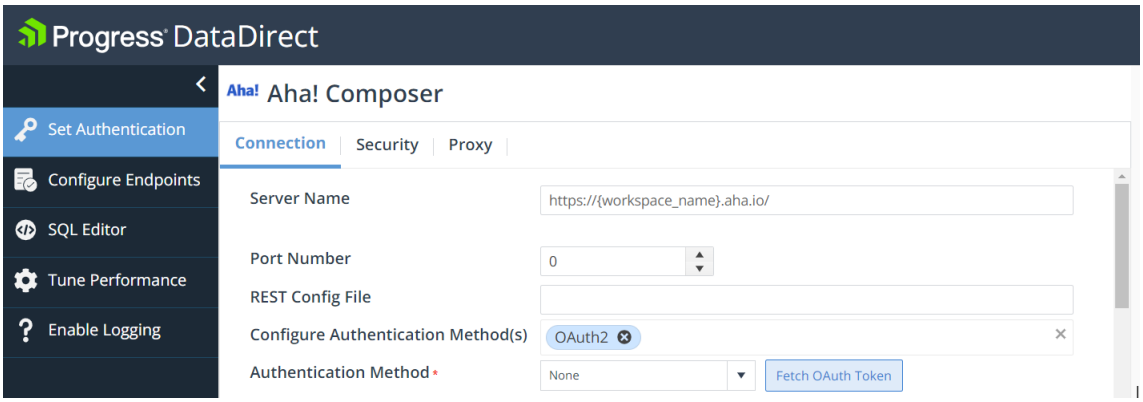


3. Select the Model for the data source to which you want to connect from the menu on the right.

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 59.

---

4. The **Connection** tab opens.



Provide values for the following fields:

- **Server 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.

5. 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.
6. 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.
7. 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.
8. To test your configuration:
   a) Click **Test Connect** or select the **SQL Editor** tab from the side menu.
   b) In the **Editor** pane, specify a query that you would like to test.
   c) Click **Execute**.

   If successful, the driver will return your results in the **Results** pane.
9. Download your model file:
   a) Select the **Configure Endpoints** tab.
   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: For subsequent connections, you will need to specify the fully qualified path to your Model file using the Config connection property. The string generated by the Autonomous REST Composer does not automatically provide a location.

10. As you provide values for properties, the Composer generates a JDBC connection string for use by your application. To use your string, on the **Set Authentication** tab, click the Copy button (). You can then 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:
    - DataDirect Configuration Manager: The DataDirect Configuration Manager allows you to test connect and execute SQL statements right out of the box.
    - Tableau: Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data.
    - DbVisualizer: DB Visualizer is a database tool that allows you to connect 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 on page 18 for details.

To begin accessing data with the driver:

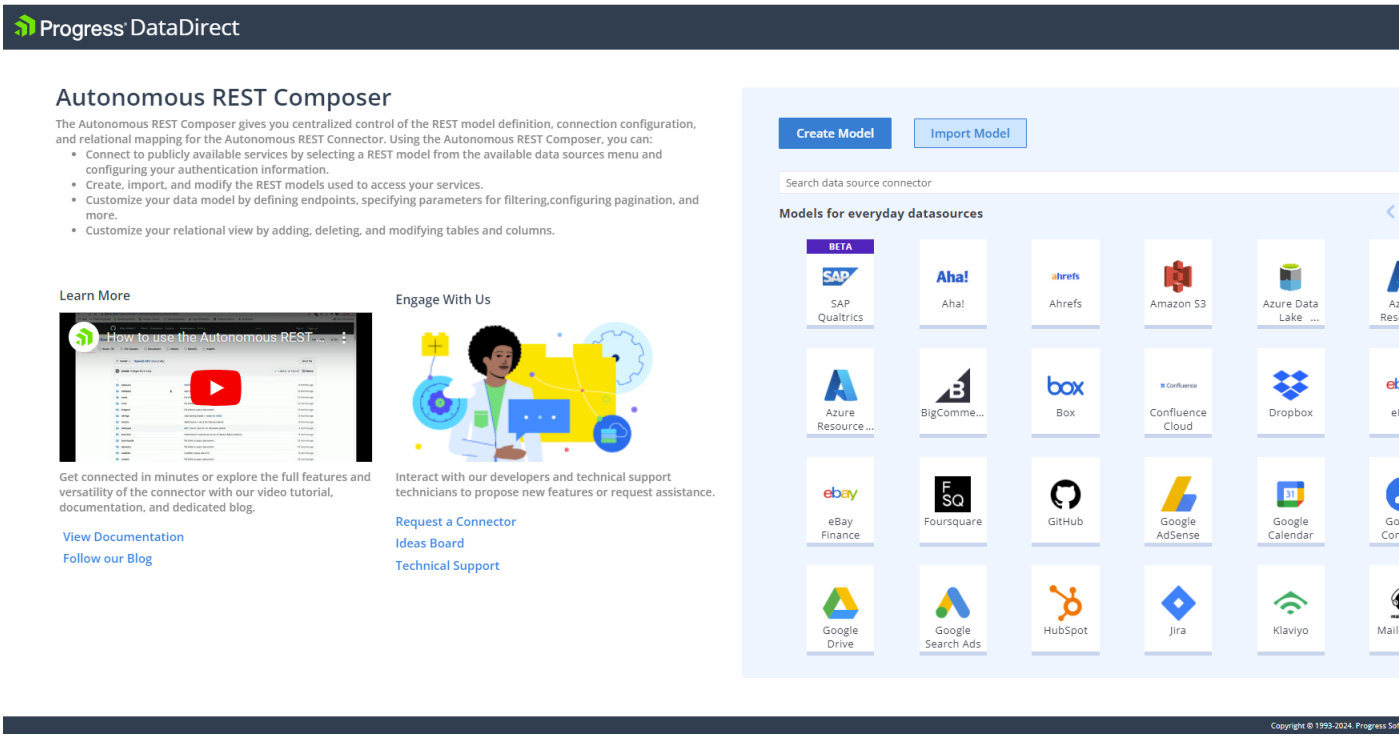
1. Install the driver:
   a) After downloading the product, unzip the installer files to a temporary directory.
   b) From the installer directory, run the appropriate installer file to start the installer.
      - **Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.exe`
      - **Non-Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.jar`
   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 JDBC Drivers Installation Guide*.
2. Open the Autonomous REST Composer by using one of the following methods:

- Select the **Autonomous REST Composer (JDBC)** 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:

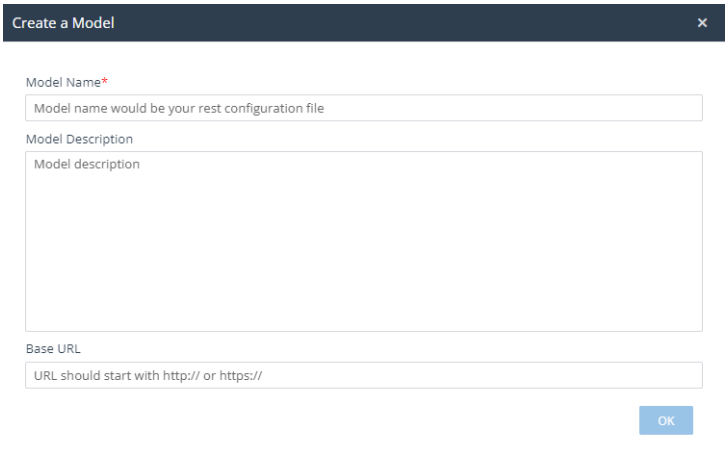
  ```
  java -jar autorest.jar --design
  ```

  By default, the `autorest.jar` file is stored in the following directory: `C:\Program Files\Progress\DataDirect\JDBC\lib\60`.



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

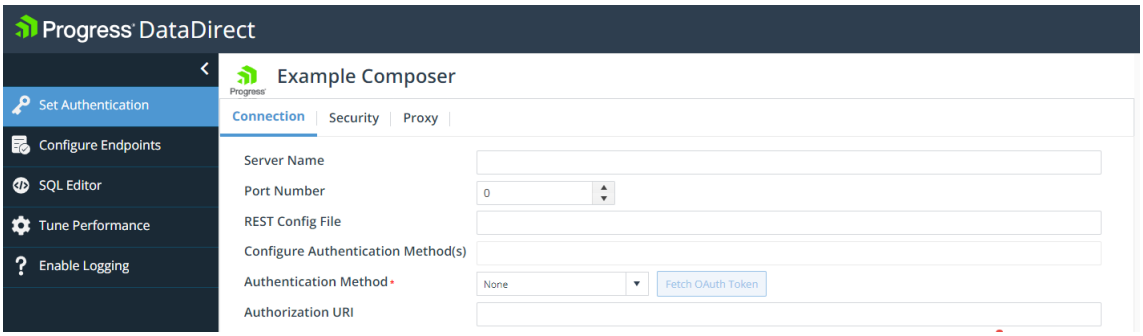
Figure 1: The Create Model Window



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.

5. The **Connection** tab opens.



Provide values for the following fields:

- **Server 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.

6. 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.

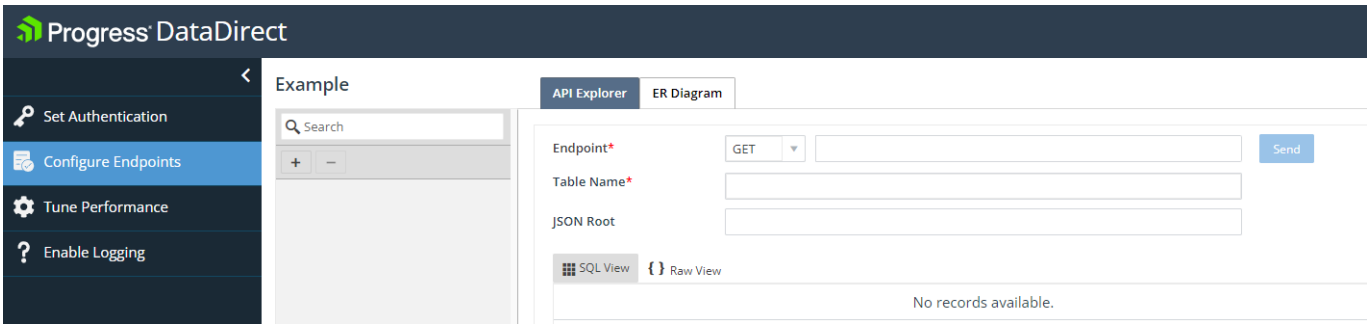
   See Authentication on page 78 for a full description of these methods.

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 84 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 67.

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

Figure 2: The Configure Endpoints tab

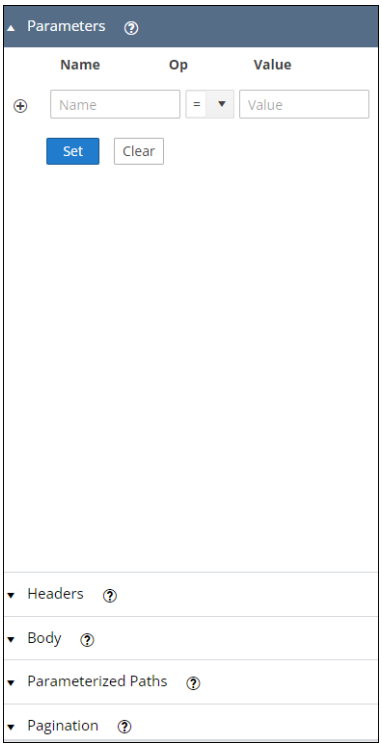


Provide the minimum required information for an endpoint to which you want to issue requests:

- 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.

8. 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 63.

Figure 3: Customization Pane



9. 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.

10. 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 197.
11. Optionally, customize your relational schema, including modifying column names, data type mapping, and primary key designation. See Customizing your schema on page 66 for details.
12. Optionally, click **Test Connect** or select the **SQL Editor** tab-->to test your model by executing SQL queries. See Testing and querying your Model on page 68 for details.
13. 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 69 for details.
14. Click **Download** to generate and download your Model file.
15. 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 property.

After creating your Model file, you are ready to configure and connect. See Installing and setting up the driver on page 25 to configure your driver for connection.

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

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, unzip the installer files to a temporary directory.
   b) From the installer directory, run the appropriate installer file to start the installer.
      - **Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.exe`
      - **Non-Windows**: `PROGRESS_DATADIRECT_JDBC_INSTALL.jar`
   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 JDBC Drivers Installation Guide*.

2. Set your system CLASSPATH to include the driver `.jar` file. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. The following examples demonstrate setting the CLASSPATH from a command line using the default installation directory.

   - **Windows Example**

     ```
     CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\autorest.jar
     ```

   - **UNIX/LINUX Example**

     ```
     CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/autorest.jar
     ```

3. Configure your driver using one of the following methods:

   - **Connection URL**: You can begin using the driver immediately by passing a connection URL with your application or tool. The following examples demonstrate the minimal connection properties required to connect with user ID and authentication:

     Basic Authentication

     ```
     jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=basic;
          Config=C:/path/to/myrest.rest;User=jsmith;Password=secret;
     ```

     OAuth 2.0 refresh token grant (Google Analytics)

     ```
     jdbc:datadirect:autorest:https://example.com/;
       AuthenticationMethod=OAuth2-RefreshToken;
       ClientID='123456789876-a1bc2de3fgh4ij567klmn8opqr9stuvw.apps.googleusercontent.com';

       ClientSecret='FaZBFRsGXTaR';Config=C:/path/to/googleanalytics.rest;
       TokenURI=https://accounts.google.com/o/oauth2/token;
     ```

     You can also generate a connection string using the Progress DataDirect REST endpoints Configuration Manager. For details, see Generating connection URLs with the Configuration Manager.

   - **Data sources**: The driver also supports connecting using JDBC data sources. A JDBC data source is a Java object, specifically a DataSource object, that defines connection information required for a JDBC driver to connect to the database. See Connecting using data sources for more information.

4. 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.
   - Performance considerations describes connection properties 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:

   - Progress DataDirect MongoDB Configuration Manager: The REST endpoints Configuration Manager is a browser-based tool that allows you to quickly generate connection URLs, test connections, and execute test queries.

- DataDirect Test: DataDirect Test allows you to test connect, execute SQL statements, and practice using the JDBC API right out of the box.
- Interactive SQL: Interactive SQL supports a command line interface that allows you to connect to a data source, execute SQL statements and retrieve results for display on a terminal. This tool provides a method of quickly testing your driver in an environment that does not support GUIs.
- Tableau: Tableau is a business intelligence software program that allows you to easily create reports and visualized representations of your data.
- DbVisualizer: DB Visualizer is a database tool that allows you to connect 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.

Driver and DataSource classes

The following are the `Driver` and `DataSource` classes used by the driver:

Driver class:

com.ddtek.jdbc.autorest.AutoRESTDriver

DataSource class:

com.ddtek.jdbcx.autorest.AutoRESTDataSource

Connection URL examples

After setting the CLASSPATH, the connection information needs to be passed in the form of a connection URL. 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.

Note:

- You can also use the DataDirect Configuration Manager tool to generate and test connection URLs. For more information, see "Generating connection URLs with the Configuration Manager."
- Connection property names are case-insensitive. For example, `Password` is the same as `password`.
- For connection properties that support string values, use the following escape sequence to specify values containing leading or trailing spaces and curly brackets: `{value}`. For example: `User={hello }` or `Password={{hello}}`.

Connection URLs for the Autonomous REST Connector can take a few different forms. Typically, the composition of the URL depends 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, note that 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):

```
jdbc:datadirect:autorest://servername;Config=model_file_path;[property=value[;...]];
```

For sessions using the Sample property:

```
jdbc:datadirect:autorest:Sample=sample_path;[property=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 authentication
- Authorization code grant authentication
- Client credentials grant authentication
- JWT bearer grant authentication
- Password grant authentication
- PKCE grant authentication
- Refresh token grant authentication
- Proxy server authentication

Basic authentication

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

```
jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=basic;
     Config=C:/path/to/myrest.rest;User=jsmith;Password=secret;
```

For more information on these properties and values, see Basic authentication on page 79.

AWS credentials authentication

This string includes the properties used to connect to a server using a Model file and AWS (Amazon Web Services) credential authentication.

```
jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=AWS;
     Config=C:/path/to/myrest.rest;AccessKey=ABCDEFGHIJKL1EXAMPLE;Region=us-east-2;
     SecretKey=aBcdeFGhiJKLM/N1OPQRS/tUvWxyzYEXAMPLEKEY";
```

For more information on these properties and values, see AWS credentials authentication on page 80.

Bearer token authentication

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

```
jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=BearerToken;
   Config=C:/path/to/myrest.rest;SecurityToken=C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx;
```

For more information on these properties and values, see Bearer token authentication on page 81.

Digest authentication

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

```
jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=basic;
    Config=C:/path/to/myrest.rest;User=jsmith;Password=secret;
```

For more information on these properties and values, see Digest authentication on page 82.

HTTP header authentication

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

```
jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=HttpHeader;
    Config=C:/path/to/myrest.rest;SecurityToken=XaBARTsLZReM;User=jsmith;
```

For more information on these properties and values, see HTTP header authentication on page 83.

URL parameter authentication

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

```
jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=UrlParameter;
    Config=C:/path/to/myrest.rest;SecurityToken=XaBARTsLZReM;User=jsmith;
```

For more information on these properties and values, see URL parameter authentication on page 84.

Access token flow authentication

This string includes the properties used to connect to a server using a Model file and OAuth 2.0 access token flow.

```
jdbc:datadirect:autorest:https://example.com/;
    AccessToken='C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx';
    AuthenticationMethod=OAuth2-AccessToken;Config=C:/path/to/myrest.rest;
```

For more information on these properties and values, see Access token flow on page 85.

Authorization code grant authentication

This string includes the properties used to connect to a server using a Model file and OAuth 2.0 authorization code grant.

```
jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-AuthorizationCode;
     ClientID='abcdefghik2lmn3o5qr67s';Config=C:/path/to/box.rest;
     OAuthCode='xyz123abc';TokenURI='https://api.box.com/oauth2/token';
```

For more information on these properties and values, see Authorization code grant on page 87.

Client credentials grant authentication

This string includes the properties used to connect to a server using a Model file and OAuth 2.0 client credentials grant.

```
jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-ClientCredentials;
  ClientID='123456789876-a1bc2de3fgh4ij567klmn8opqr9stuvw.apps.googleusercontent.com';
  ClientSecret='FaZBFRsGXTaR';Config=C:/path/to/googleanalytics.rest;
  TokenURI=https://accounts.google.com/o/oauth2/token;
```

For more information on these properties and values, see Client credentials grant on page 88.

JWT bearer grant

This string includes the properties used to connect to a server using a Model file and OAuth 2.0 JWT bearer grant.

```
jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-JWTBearer;
     ClaimsIssuer='1a2b3c4d5e_6f7g8h9g';ClaimsSubject=jsmith@example.com;
     Config=C:/path/to/salesforce.rest;JWTCertStore=jwtcert.jks;
     JWTCertPassword=secret;JWTCertAlias=myAlias;
```

For more information on these properties and values, see JWT bearer grant on page 92.

Password grant authentication

This string includes the properties used to connect to a server using a Model file and OAuth 2.0 password grant.

```
jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-Password;
    ClientID='123456789876-a1bc2de3fgh4ij567klmn8opqr9stuvw.apps.googleusercontent.com';

     ClientSecret='FaZBFRsGXTaR';Config=C:/path/to/googleanalytics.rest;
     TokenURI=https://accounts.google.com/o/oauth2/token;
```

For more information on these properties and values, see Password grant on page 94.

PKCE grant authentication

This string includes the properties used to connect to a server using a Model file and OAuth 2.0 PKCE grant.

```
jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-PKCE;
     AuthURI=https://accounts.spotify.com/authorize;ClientID='abcdefghik2lmn3o5qr67s';
     ClientSecret=FaZBFRsGXTaR;Config=C:/path/to/spotify.rest;
    RedirectURI=https://localhost:8080;TokenURI='https://accounts.spotify.com/api/token';
```

For more information on these properties and values, see PKCE grant on page 95.

Refresh token grant authentication

This string includes the properties used to connect to a server using a Model file and OAuth 2.0 refresh token grant.

```
jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-RefreshToken;
    ClientID='123456789876-a1bc2de3fgh4ij567klmn8opqr9stuvw.apps.googleusercontent.com';

     ClientSecret='FaZBFRsGXTaR';Config=C:/path/to/googleanalytics.rest;
     TokenURI=https://accounts.google.com/o/oauth2/token;
```

For more information on these properties and values, see Refresh token grant on page 97.

Proxy server authentication

This string includes the properties used to connect using a Model file through a proxy server with basic authentication.

```
jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=basic;
     Config=C:/path/to/myrest.rest;ProxyHost=proxy_host;
     ProxyPassword=proxy_password;ProxyPort=proxy_port;
     ProxyUser=proxy_user;User=jsmith;Password=secret;
     [property=value[;...]];
```

See also

Connecting using the JDBC Driver Manager on page 70

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 QualifyNormalizedNames property 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 "QualifyNormalizedNames" connection property 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

QualifyNormalizedNames on page 158
Generating a Model file with the Autonomous REST Composer on page 59
Requests with custom HTTP headers on page 210

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

Table 1: 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 2: 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 3: 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 39
Customizing your schema on page 66
Table on page 172

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 8061 format. The map may contain values with and without timezones in the same map.
  - Timestamps using the ISO 8061 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"},
}
```

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 4: SEASON_RESULTS

KEY (PK)	ATTENDANCE	OPPONENT	RESULT
1979-10-31	12080	Wildcats	loss
1979-12-06	34000	Mustangs	loss
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 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 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 5: 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 6: 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 7: 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 39
Customizing your schema on page 66
Table on page 172

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 8: 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 39
Customizing your schema on page 66
Table on page 172

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.

Note: 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 and "Customizing your schema" to use the Autonomous REST Composer.

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 219
Customizing your schema on page 66
Reviewing the status of your endpoints on page 69

Data types

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

See "getTypeInfo()" for getTypeInfo() results of data types supported by the driver.

Table 9: Autonomous REST Connector Data Types

Autonomous REST Connector Data Type	JDBC Data Type
BigInt	BIGINT
Binary	BINARY
Bit	BIT
Boolean	BOOLEAN
Char	CHAR

Autonomous REST Connector Data Type	JDBC Data Type
Date	DATE
Decimal	DECIMAL
Double	DOUBLE
Float	FLOAT
GUID	CHAR
Integer	INTEGER
JSON	VARCHAR
LongVarBinary	LONGVARBINARY
LongVarChar	LONGVARCHAR
NVarChar	NVARCHAR
SmallInt	SMALLINT
Time	TIME
Timestamp	TIMESTAMP
TinyInt	TINYINT
VarBinary	VARBINARY
VarChar	VARCHAR

getTypeInfo()

The DatabaseMetaData.getTypeInfo() method returns information about data types. The following table provides getTypeInfo() results for supported data types.

Table 10: getTypeInfo() Results

TYPE_NAME = BigInt AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -5 (BIGINT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = BigInt MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 19 SEARCHABLE = 3 SQL_DATA_TYPE = 25 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = Binary AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -2 (BINARY) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Binary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32767 SEARCHABLE = 3 SQL_DATA_TYPE = 60 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Bit AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -7 (BIT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Bit MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = 14 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = Boolean AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 16 (BOOLEAN) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Boolean MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 1 SEARCHABLE = 3 SQL_DATA_TYPE = 16 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Char AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Char MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 255 SEARCHABLE = 3 SQL_DATA_TYPE = 1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Date AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 91 (DATE) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = DATE ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Date MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = 91 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = Decimal AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 3 (DECIMAL) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Decimal MAXIMUM_SCALE = 1000	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 1000 SEARCHABLE = 3 SQL_DATA_TYPE = 3 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = Double AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 8 (DOUBLE) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Double MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 53 SEARCHABLE = 3 SQL_DATA_TYPE = 8 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = Float AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 6 (FLOAT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Float MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = 2 PRECISION = 24 SEARCHABLE = 3 SQL_DATA_TYPE = 6 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE

TYPE_NAME = GUID AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 1 (CHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = GUID MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 36 SEARCHABLE = 3 SQL_DATA_TYPE = 1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Integer AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 4 (INTEGER) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = Integer MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 10 SEARCHABLE = 3 SQL_DATA_TYPE = 4 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = JSON AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = JSON MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 3 SQL_DATA_TYPE = 12 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = LongVarBinary AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -4 (LONGVARBINARY) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LongVarBinary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 0 SQL_DATA_TYPE = -4 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = LongVarChar AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = -1 (LONGVARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = LongVarChar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 0 SQL_DATA_TYPE = -1 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = NVarChar AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = -9 (NVARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = NVarChar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32767 SEARCHABLE = 3 SQL_DATA_TYPE = -9 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = SmallInt AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 5 (SMALLINT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = SmallInt MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 5 SEARCHABLE = 3 SQL_DATA_TYPE = 5 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = Time AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 92 (TIME) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = TIME ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Time MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 12 SEARCHABLE = 3 SQL_DATA_TYPE = 92 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = Timestamp AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = 93 (TIMESTAMP) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = TIMESTAMP ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = Timestamp MAXIMUM_SCALE = 9	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 23 SEARCHABLE = 3 SQL_DATA_TYPE = 93 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = TinyInt AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -6 (TINYINT) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = NULL LITERAL_SUFFIX = NULL LOCAL_TYPE_NAME = TinyInt MAXIMUM_SCALE = 0	MINIMUM_SCALE = 0 NULLABLE = 1 NUM_PREC_RADIX = 10 PRECISION = 3 SEARCHABLE = 3 SQL_DATA_TYPE = -6 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = FALSE
TYPE_NAME = VarBinary AUTO_INCREMENT = FALSE CASE_SENSITIVE = FALSE CREATE_PARAMS = NULL DATA_TYPE = -3 (VARBINARY) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = X' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = VarBinary MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 16777215 SEARCHABLE = 3 SQL_DATA_TYPE = 61 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL
TYPE_NAME = VarChar AUTO_INCREMENT = FALSE CASE_SENSITIVE = TRUE CREATE_PARAMS = NULL DATA_TYPE = 12 (VARCHAR) FIXED_PREC_SCALE = FALSE LITERAL_PREFIX = ' LITERAL_SUFFIX = ' LOCAL_TYPE_NAME = VarChar MAXIMUM_SCALE = NULL	MINIMUM_SCALE = NULL NULLABLE = 1 NUM_PREC_RADIX = NULL PRECISION = 32767 SEARCHABLE = 3 SQL_DATA_TYPE = 12 SQL_DATETIME_SUB = NULL UNSIGNED_ATTRIBUTE = NULL

SQL escape sequences

The driver supports the following SQL escape sequences.

- Date, Time, and Timestamp Escape Sequences
- Scalar Functions
- Outer Join Escape Sequences
- LIKE Escape Character Sequence for Wildcards

Refer to "SQL escape sequences" in the *Progress DataDirect for JDBC Drivers Reference* for information about SQL escape sequences.

Supported scalar functions

The driver supports the scalar functions in the following table. Note that your database system may not support all these functions. Refer to the documentation for your database system to find out which functions are supported by your database.

In addition, you can also determine the supported scalar functions by using DatabaseMetaData methods.

You can use scalar functions in SQL statements with the following syntax:

`{fn scalar-function}`

where:

scalar-function

is a scalar function supported by the drivers, as listed in the following table.

Example:

```
SELECT id, name FROM emp WHERE name LIKE {fn UCASE('Smith')}
```

Refer to "Scalar functions" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

Table 11: Supported Scalar Functions

String Functions	Numeric Functions	Timedate Functions	System Functions
ASCII	ABS	CURDATE	CURSESSIONID
BIT_LENGTH	ACOS	CURRENT_DATE	DATABASE
CHAR	ASIN	CURRENT_TIME	IDENTITY
CHAR_LENGTH	ATAN	CURRENT_TIMESTAMP	USER
CHARACTER_LENGTH	ATAN2	CURTIME	
CONCAT	BITAND	DATEDIFF	

String Functions	Numeric Functions	Timedate Functions	System Functions
DIFFERENCE	BITOR	DATE_ADD	
HEXTORAW	BITXOR	DATE_SUB	
INSERT	CEILING	DAY	
LCASE	COS	DAYNAME	
LEFT	COT	DAYOFMONTH	
LENGTH	DEGREES	DAYOFWEEK	
LOCATE	EXP	DAYOFYEAR	
LOCATE_2	FLOOR	EXTRACT	
LOWER	LOG	HOUR	
LTRIM	LOG10	MINUTE	
OCTET_LENGTH	MOD	MONTH	
RAWTOHEX	PI	MONTHNAME	
REPEAT	POWER	NOW	
REPLACE	RADIANS	QUARTER	
RIGHT	RAND	SECOND	
RTRIM	ROUND	SECONDS_SINCE_MIDNIGHT	
SOUNDEX	ROUNDMAGIC	TIMESTAMPADD	
SPACE	SIGN	TIMESTAMPDIFF	
SUBSTR	SIN	TO_CHAR	
SUBSTRING	SQRT	WEEK	
UCASE	TAN	YEAR	
UPPER	TRUNCATE		

Driver specifications

This section describes the general functionality supported by the driver.

- **Unicode support**: Multilingual JDBC applications can be developed on any operating system using the driver to access both Unicode and non-Unicode enabled databases. Internally, Java applications use UTF-16 Unicode encoding for string data. When fetching data, the driver automatically performs the conversion from the character encoding used by the database to UTF-16. Similarly, when inserting or updating data in the database, the driver automatically converts UTF-16 encoding to the character encoding used by the database.

  The JDBC API provides mechanisms for retrieving and storing character data encoded as Unicode (UTF-16) or ASCII. Additionally, the Java String object contains methods for converting UTF-16 encoding of string data to or from many popular character encodings.

- **Isolation and lock levels**: Because transactions are not supported, the driver supports only the isolation level 0 (read uncommitted).
- **Error handling**: The driver reports errors to the application by throwing SQLExceptions. Each SQLException contains the following information:
  - Description of the probable cause of the error, prefixed by the component that generated the error
  - Native error code (if applicable)
  - String containing the XOPEN SQLstate

DataDirect tools

Progress DataDirect for JDBC drivers install the set of tools described in this section. For detailed instructions on using these tools, refer to the corresponding topics in the *Progress DataDirect for JDBC Drivers Reference*.

- DataDirect Test allows you to test your JDBC driver and learn the JDBC API.
- DataDirect Connection Pool Manager allows you to pool connections when accessing databases. When your applications use connection pooling, connections are reused rather than created each time a connection is requested. Because establishing a connection is among the most costly operations an application may perform, using Connection Pool Manager to implement connection pooling can significantly improve performance.
- Statement Pool Monitor loads statements into and remove statements from the statement pool as well as generate information to help you troubleshoot statement pooling performance.
- DataDirect Spy logs detailed information about calls your driver makes that can be used for troubleshooting.

Troubleshooting

The *Progress DataDirect for JDBC Drivers Reference* provides information on troubleshooting problems should they occur. Refer to the "Troubleshooting" section in the *Reference* for details.

Additional information

In addition to the content provided in this guide, the documentation set also contains detailed conceptual and reference information that applies to all the drivers. For more information in these topics, refer the *Progress DataDirect for JDBC Drivers Reference* or use the links below to view some common topics:

- "JDBC support" describes support for JDBC interfaces and methods for the Progress DataDirect for JDBC drivers.
- "JDBC extensions" describes the JDBC extensions provided by the com.ddtek.jdbc.extensions package.
- "SQL escape sequences for JDBC" provides an overview of SQL escape sequences for JDBC. In addition, it documents the scalar functions that you use in SQL statements.
- "Security best practices for JDBC applications" describes the security best practices you should employ when developing and deploying your application with the driver.

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.

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."

For details, see the following topics:

- Interactive SQL
- Tableau
- DbVisualizer

Interactive SQL

After you have installed your driver, you can use the driver to access your data with the Interactive SQL tool. Interactive SQL supports a command line interface that allows you to connect to a data source, execute SQL statements and retrieve results for display on a terminal.

To execute commands with Interactive SQL:

1. Start the ISQL tool. From a command line, enter the following:

   ```
   java -jar autorest.jar --isql
   ```

2. Enter connection properties one at a time by typing *property=value*, then pressing **Enter**. For example, to configure the ServerName property:

   ```
   ServerName=myserver
   ```

3. After specifying values for your properties, type `connect`, then press **Enter**. If successful, the tool will return a confirmation message.

   Note: If you are unable to connect, you can review the URL by entering the `SHOW URL` command.

4. At the `ISQL>` prompt, issue a SQL command to query or modify the data source; then, press **Enter**. For example:

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

   Note: SQL commands must be terminated by a semi-colon.

   Note: In addition to SQL commands, the tool supports a set of proprietary commands. Type `Help` at the prompt for a list of supported commands and syntax.

   The results of the command are displayed in the terminal.

5. After you are finished executing queries and commands, you can disconnect from the data source by typing the following; then, pressing **Enter**:

   ```
   DISCONNECT
   ```

6. To end the session, type `exit`; then, press **ENTER**.

Tableau

After you have installed your driver and defined it on the CLASSPATH, 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 `\lib\`*`xx`* subdirectory of the Progress DataDirect installation directory; then, locate the `jar` file for your driver:

   ```
   autorest.jar
   ```

2. Copy the `.jar` file for your driver into the following directory:

   Windows: `C:\Program Files\Tableau\Drivers`

   Linux: `/opt/tableau/tableau_driver/jdbc`

3. Open Tableau. From the **Connect** menu, select **Other Databases (JDBC)**.
4. In the **Other Databases (JDBC)** dialog, provide values for the following fields; then, click **Sign In**.
   - **URL**: Copy and paste your connection URL into this field. The following example demonstrates how to connect using the Sample property with no authentication.

     ```
     jdbc:datadirect:autorest:Sample='https://example.com/countries/';
     ```

   - **Dialect**: Select **SQL92** (the default) from the drop-down box.
5. The **Data Source** window appears. In the **Schema** field, select the schema for the service you want to use.
6. In the **Table** field, the tables stored in the selected schema are now exposed and available for selection.

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

DbVisualizer

After you have installed your driver and defined it on the CLASSPATH, you can use the driver to access your data with the third-party DbVisualizer tool. The following topics guide you through using DbVisualizer to add your driver, connect, and execute SQL statements.

Adding a driver

To add a driver with DbVisualizer:

1. Open DbVisualizer.
2. From the menu, select **Tools**>**Driver Manager**. The Driver Manager window opens.
3. From the Driver Manager menu, select **Driver**>**Create Driver**.
4. Click the  button to navigate to the location of the driver jar file; then, click **OK**. The following are the default locations for the driver:

   Windows

   ```
   C:\Program Files\Progress\DataDirect\JDBC\lib\60\autorest.jar
   ```

   Linux

   ```
   /opt/Progress/DataDirect/JDBC/lib/60/autorest.jar
   ```

5. Provide values for the following fields; then, close the Driver Manager window.
   - **Name**: Type an alias for your driver. For example:

     ```
     Autonomous REST Connector
     ```

   - **URL Format**: Optionally, specify the format of the connection string for your driver. For example:

     ```
     jdbc:datadirect:autorest:Config=C:/path/to/myrest.rest;
     ```

- **Driver Class**: From the drop down menu, select the driver class for your driver. For example:

  ```
  com.ddtek.jdbcx.autorest.AutoRESTDataSource
  ```

You can now use your driver with DbVisualizer. Proceed to "Connecting and executing SQL statements" for information on connecting and executing SQL statements.

Connecting and executing SQL statements

To use the driver to access data with DbVisualizer:

1. Open DbVisualizer.
2. From the menu, select **Database**>**New Connection**. When prompted to use the Connection Wizard, click **OK**.
3. Provide the following information when prompted; then, click **Next** to proceed:
   - **Connection alias**: Type the name to be used when referring to this connection.
   - **Driver**: Select the alias that you provided for your driver from the drop-down menu.
4. Provide values for the following fields; then, click **Finish**.
   - **Database URL**: Copy and paste your connection URL into this field. The following example demonstrates how to connect using the Sample property with no authentication.

     Note: You can also generate connection strings using the Autonomous REST Connector Configuration Manager. For more information, see Generating connection URLs with the Configuration Manager on page 71.

     ```
     jdbc:datadirect:autorest:Sample='https://example.com/countries/';
     ```
5. To execute SQL statements, select **SQL Commander**>**New SQL Commander**. A SQL Commander tab opens.
6. Select values for the following fields:
   - **Database Connection**: Select connection alias you provided for the connection from the drop-down menu.
   - **Schema**: Select the schema you want to execute queries against from the drop-down menu.
7. In the SQL Commander tab, enter SQL commands you want to execute; then select **SQL Commander**>**Execute**. For example:

   To select all of the rows from the `INFORMATION_SCHEMA.SYSTEM_TABLES` table:

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

You have successfully accessed your data with DbVisualizer.

3

Configuring and connecting

This section provides information on how to connect to your data store using either the JDBC Driver Manager or DataDirect JDBC data sources, as well as information on how to implement and use functionality supported by the driver.

After the driver has been installed and defined on your classpath, you can connect from your application to your data in either of the following ways.

- Using the JDBC `DriverManager` by specifying the connection URL in the `DriverManager.getConnection()` method.
- Creating a JDBC data source that can be accessed through the Java Naming Directory Interface (JNDI).

For details, see the following topics:

Setting the classpath

The driver must be defined on your CLASSPATH before you can connect. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. If the driver is not defined on your CLASSPATH, you will receive a `class not found` exception when trying to load the driver. Set your system CLASSPATH to include the driver jar file as shown, where *`install_dir`* is the path to your product installation directory.

```
install_dir/lib/60/autorest.jar
```

Windows Example

```
CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC\lib\60\autorest.jar
```

UNIX Example

```
CLASSPATH=.:/opt/Progress/DataDirect/JDBC/lib/60/autorest.jar
```

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 Config (REST Config File) property 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 "Example 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

Config on page 133
Generating a Model file with the Autonomous REST Composer on page 59
Model file syntax on page 181
Example Model file on page 248
Getting started using prebuilt Model files on page 18

Generating a Model file with the Autonomous REST Composer

The Autonomous REST Composer allows you to generate and edit Model files, instead of manually creating a file. The primary purpose of the Model file is to define endpoints 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. See "Configuring the relational map" for an overview of the Model file.

To generate your Model file:

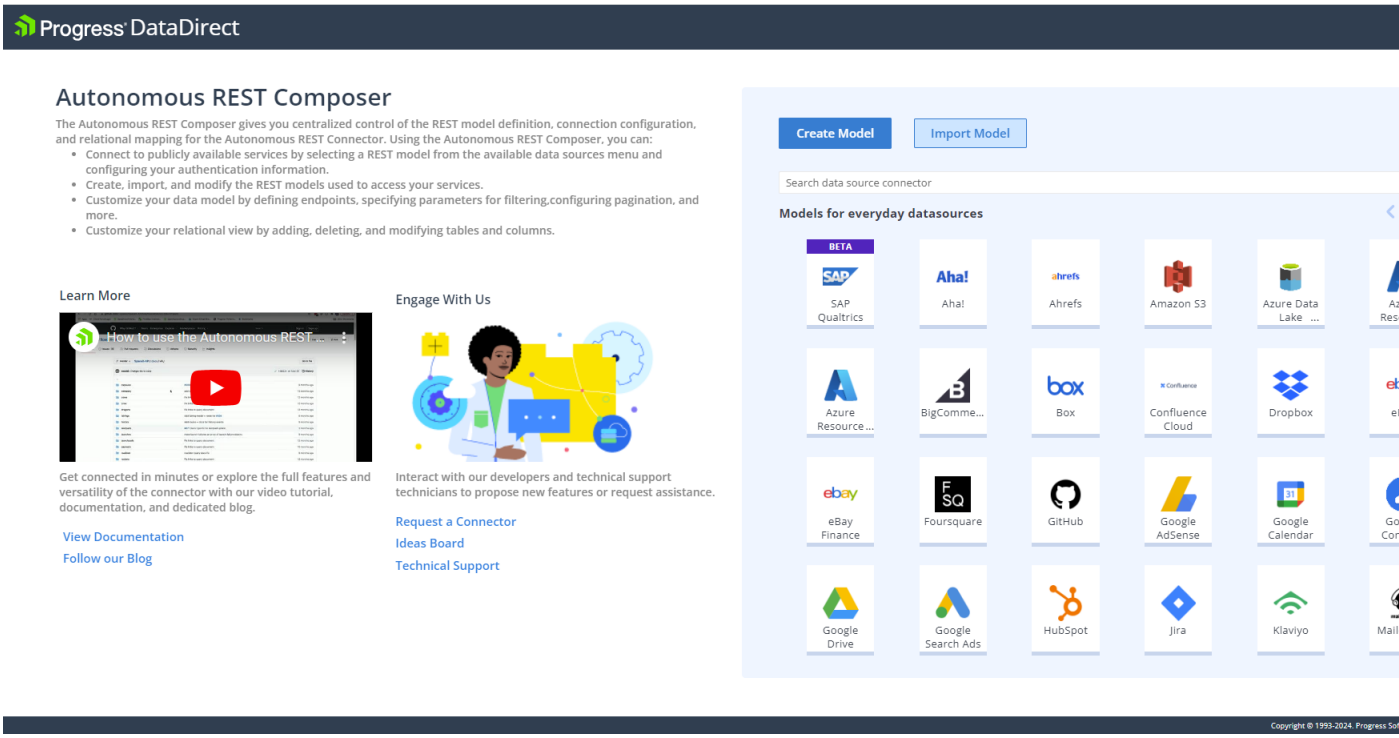
1. Open the Autonomous REST Composer by using one of the following methods:
   - Select the **Autonomous REST Composer (JDBC)** icon from your desktop or the Windows Start menu.
   - From a command line, navigate to the directory containing the `autorest.jar` file and execute the following command:

     ```
     java -jar autorest.jar --design
     ```

     By default, the `autorest.jar` file is stored in the following directory: `C:\Program Files\Progress\DataDirect\JDBC\lib\60`.

The Autonomous REST Composer opens in your default web browser.

Figure 4: Welcome screen for Autonomous REST Composer



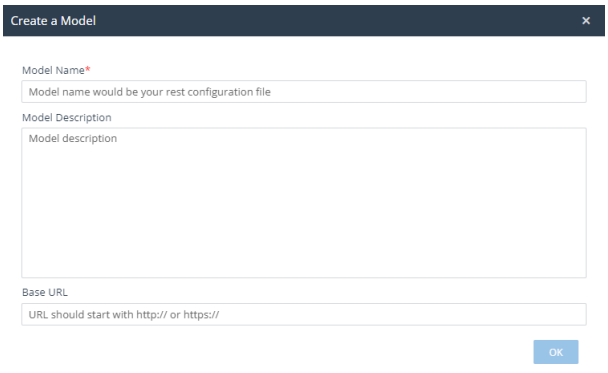
2. Select **Create a Model**.

Note: If you are accessing a publicly available data source, refer to the library of prebuilt Model files for everyday data sources. 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 on page 18 for details.

---

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

Figure 5: The Create Model Window



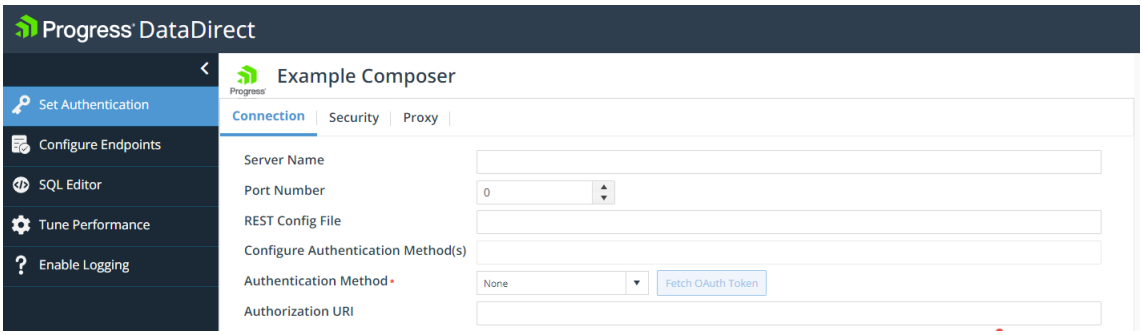
Complete the following fields to create a new project; then, click **OK**:

- **Model Name**: The name of your project and 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 Autonomous REST Composer opens to the Connection tab.

Figure 6: The Connection tab



4. Select the Authentication Method used by your endpoints; then, provide values for the appropriate fields. Note that not all of the fields exposed are required for all services. See Authentication on page 78 for a full description of these methods.

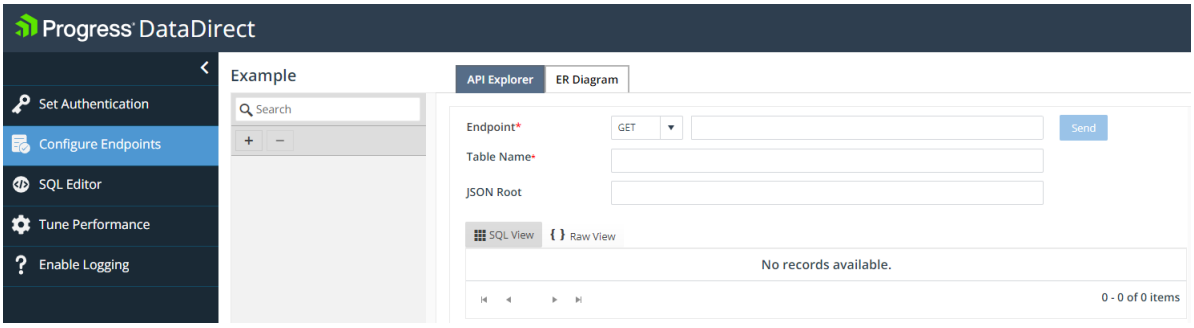
Note: Authentication properties specified through the Autonomous REST Composer are not persisted in the Model file. To share authentication settings among all connections using the file, you must manually update the Model file. See OAuth 2.0 authentication on page 186 for details.

Note: Custom authentication properties must be specified manually in the Model file. For details, see Custom authentication requests on page 188.

---

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

Figure 7: The Configure Endpoints tab

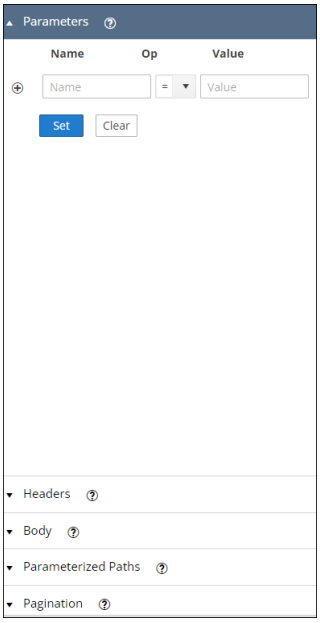


Provide the minimum required information for an endpoint to which you want to issue requests:

- From the **Endpoint** drop-down menu, select the type of request to issue against your endpoint.
- In the **Endpoint** field, type the path portion of your endpoint after the base URL. 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.
- 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 63.

   Figure 8: 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 197.
9. Optionally, customize your relational schema, including modifying column names, data type mapping, and primary key designation. See Customizing your schema on page 66 for details.
10. Optionally, click **Test Connect** or select the **SQL Editor** tab to test your model by executing SQL queries. See Testing and querying your Model on page 68 for details.
11. Click **Download** to generate and download your Model file.
12. 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 Config connection property.

After creating your Model file, you are ready to configure and connect. You can edit your Model file later by selecting **Import a Model** from the Hub window when starting the Configuration Manger.

Note: The Model file generated by the Autonomous REST Composer 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 by generating the file through the Autonomous REST Composer. If you need to configure features or functionality not supported through the Autonomous REST Composer, you can manually edit your generated file using a text editor. See Model file syntax on page 181 for details.

See also

Configuring the relational map on page 58
URL-encoded values on page 246

Sampling REST endpoints

The following sections guide you through sampling different kinds of requests with the 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 valid URL-encoded 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 valid URL-encoded 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 valid URL-encoded 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 **Parameterized 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 the body of 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 valid URL-encoded 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 valid URL-encoded 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

If you only want to return results for an embedded object:

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 246

Parameter variables

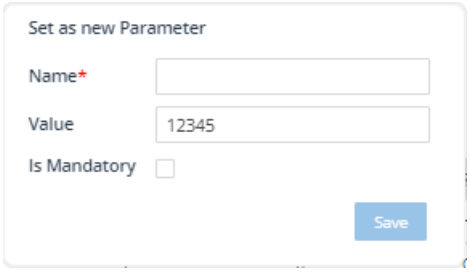
When creating or editing a REST model file, the parameter values you define in an endpoint might not apply to all users of the service. For example, using a parameter that filters results by your user ID or project might retrieve data relevant only to you. To facilitate sharing REST model files, the 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 9: 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.

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 188 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 181 and Custom authentication requests on page 188 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 input REST 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 CustomAuthParams on page 135 for details.

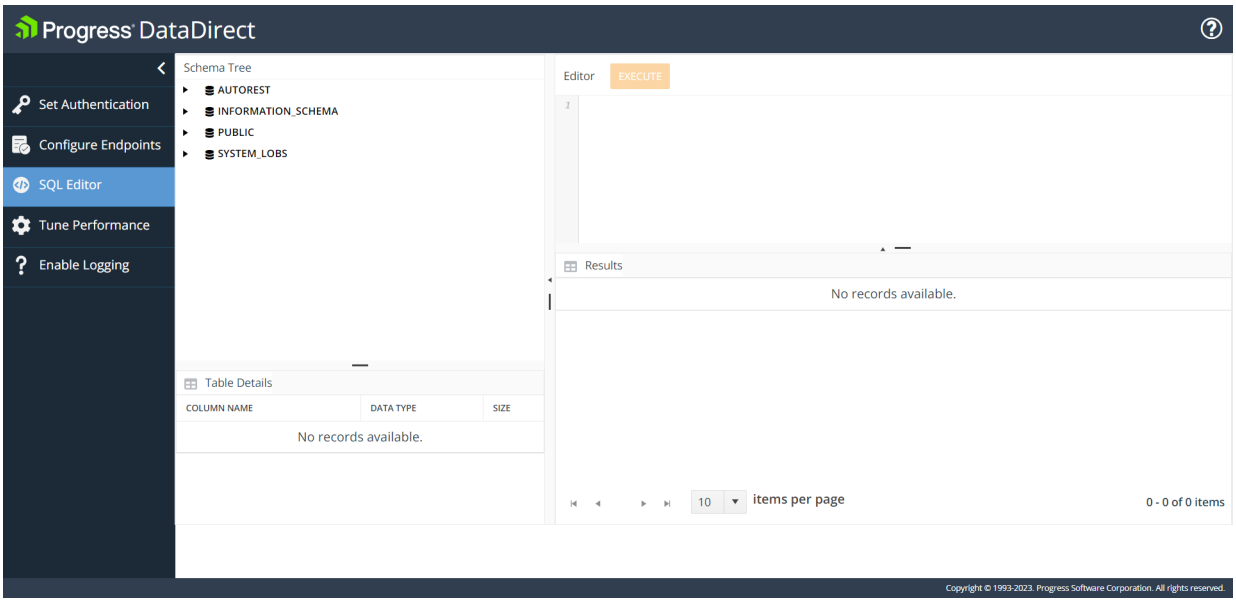
This completes configuring custom authentication using the Autonomous REST Composer.

Testing and querying your Model

The Autonomous REST Composer includes a SQL Editor that allows you to view your model and test it by executing SQL queries.

After you have configured one or more endpoints, you can take the following steps to view your schema and execute queries against it.

1. From the Autonomous REST Composer, open the SQL Editor by using one of the following methods:
   - Select the **SQL Editor** tab from the side menu.
   - Click **Test Connect** on any of the tabs.



2. To view tables and objects in your schemas, expand the schema you want to view from the **Schema Tree** pane.
3. To view the details of a table or object, select the table or object from the **Schema Tree**. Details such as column names and data types appear in the **Table Details** pane.
4. To query your Model, enter a SQL query in the **Editor** pane and then click **EXECUTE** to run the query.

   Note: You can double-click on table names to quickly add a basic SELECT statement to the **Editor** pane.

   The results of the query are displayed in the **Results** section along with the status of the query execution. The maximum number of rows displayed per query is 200.
5. Optionally, you can review the status of your endpoints by querying the `INFORMATION_SCHEMA.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.

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.

Note that, in certain situations, you might need to set `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 SamplingFailureTolerance, 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 SamplingFailureTolerance to audit your complete set of endpoints.

Connecting using the JDBC Driver Manager

One way to connect to a service is through the JDBC DriverManager using the `DriverManager.getConnection()` method. As the following example shows, this method specifies a string containing a connection URL.

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:autorest:Config=C:/path/to/example.rest;");
```

Passing the connection URL

After setting the CLASSPATH, the required connection information needs to be passed in the form of a connection URL. The form of the connection URL differs depending on whether you are using a Model file.

Connection URL Syntax

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

```
("jdbc:datadirect:autorest://servername;Config=model_file_path;[property=value[;...]];")
```

For sessions using the Sample property:

```
("jdbc:datadirect:autorest:Sample=sample_path;[property=value[;...]];")
```

where:

servername

optionally, specifies the host name portion of the HTTP endpoint to which you send requests. Specify this value only if you want to define endpoints without the web server address in the REST config file.

model_file_path

specifies the name and location of the Model file that contains a list of endpoints to sample, PUSH request definitions, and configuration information. See "Creating a Model file" for details.

property=value

specifies connection property settings. Multiple properties are separated by a semi-colon.

sample_path

specifies the endpoint the sample when not using a Model file.

Connection URL Example

For sessions using a Model file:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:autorest:https://example.com/;Config=C:\path\to\myrest.rest;");
```

For sessions using the Sample property:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:autorest:Sample='https://example.com//countries/get/all';");
```

See also

Config on page 133
Sample on page 164

Generating connection URLs with the Configuration Manager

Note: You can also use the Autonomous REST Composer (administrator view) to generate URLs 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 driver includes a browser-based tool, Progress DataDirect Autonomous REST Connector Configuration Manager, that allows you to generate connection URLs, test connections, and execute test queries. This section will guide you through generating and testing a connection URL that can be used by your application.

To generate a connection URL:

1. Open the Autonomous REST Connector Configuration Manager by double-clicking on the driver jar file. Or, in a command line, navigate to the directory containing your driver jar file; then execute the following command:

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

   The Autonomous REST Connector Configuration Manger opens in your default web browser.

2. From the browser window, provide values of the connection properties you want to configure in the corresponding fields. A connection URL will generate in the Connection String field as you provide settings. To view more properties, select the tabs at the top of the page. See "Connection URL examples" for a list of required properties and properties used for different configurations.

Note: If you do not specify a value for an optional property, the property will be omitted from the string and the default value will be used.

3. Optionally, you can manually edit your string by clicking the Edit button ( ).
4. At any point during the process, you can click **Test Connect** to attempt to connect to the service using the string generated in the Connection String field. In the **Test Connection** window:
   a) Provide values for any fields required by your service.
   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.

Note: The information you enter in the logon dialog box during a test connect is not saved.

5. To use your string, click the Copy button ( ) and paste the string to a location that can be used by your application.
6. Optionally, click **Save** to store your connection string for later use.

See also

Getting started using prebuilt Model files on page 18
Generating a Model file with the Autonomous REST Composer on page 59

Testing connections and queries

You can quickly test a connection string 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 double-clicking on the driver jar file. Or, in a command line, navigate to the directory containing your driver jar file; then, execute the following command:

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

   The Autonomous REST Connector Configuration Manager opens in your default web browser.

2. Provide a connection string to test using one of the following methods:

   - Entering a string: Click the Edit button ( ); then, paste your string into the Connection String field. If you prefer, you can also type a string directly into this field.
   - Generating a string: Provide values of the connection properties you want to configure into the corresponding fields. The Autonomous REST Connector Configuration Manager will generate a string in the Connection String field 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 property values for any fields required by your service.
5. 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.

Connecting using data sources

A *JDBC data source* is a Java object, specifically a `DataSource` object, that defines connection information required for a JDBC driver to connect to the database. Each JDBC driver vendor provides their own data source implementation for this purpose. A Progress DataDirect data source is Progress DataDirect's implementation of a `DataSource` object that provides the connection information needed for the driver to connect to a database.

Because data sources work with the Java Naming Directory Interface (JNDI) naming service, data sources can be created and managed separately from the applications that use them. Because the connection information is defined outside of the application, the effort to reconfigure your infrastructure when a change is made is minimized. For example, if the database is moved to another database server, the administrator need only change the relevant properties of the `DataSource` object. The applications using the database do not need to change because they only refer to the name of the data source.

How data sources are implemented

Data sources are implemented through a `DataSource` class. A data source class implements the following interfaces.

- `javax.sql.DataSource`
- `javax.sql.ConnectionPoolDataSource` (allows applications to use connection pooling)

Refer to "Connection Pool Manager" in the *Progress DataDirect for JDBC Drivers Reference* for more information.

See also

Driver and DataSource classes on page 27

Creating data sources

The following example files provide details on creating and using Progress DataDirect data sources with the Java Naming Directory Interface (JNDI), where *`install_dir`* is the product installation directory.

- *`install_dir`*`/Examples/JNDI/JNDI_LDAP_Example.java` can be used to create a JDBC data source and save it in your LDAP directory using the JNDI Provider for LDAP.
- *`install_dir`*`/Examples/JNDI/JNDI_FILESYSTEM_Example.java` can be used to create a JDBC data source and save it in your local file system using the File System JNDI Provider.

See "Example data source" for an example data source definition for the example files.

To connect using a JNDI data source, the driver needs to access a JNDI data store to persist the data source information. For a JNDI file system implementation, you must download the File System Service Provider from the Oracle Technology Network Java SE Support downloads page, unzip the files to an appropriate location, and add the `fscontext.jar` and `providerutil.jar` files to your CLASSPATH. These steps are not required for LDAP implementations because the LDAP Service Provider is included with supported versions of Java SE.

Example data source

To configure a data source using the example files, you will need to create a data source definition. The content required to create a data source definition is divided into three sections.

First, you will need to import the data source class. For example:

```
import com.ddtek.jdbcx.autorest.AutoRESTSource;
```

Next, you will need to set the values and define the data source. For example, the following definition contains the minimum properties required for a connection using a Model file with no authentication.

Note:

- Setting the password using a data source is generally not recommended. The data source persists all properties, including the Password property, in clear text.
- In a JDBC data source, string values must be enclosed in double quotation marks, for example, `setUser("abc@defcorp.com")`.

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Connector Data Source");
mds.setConfig("C:/path/to/myrest.rest");
```

Finally, you will need to configure the example application to print out the data source attributes. Note that this code is specific to the driver and should only be used in the example application. For example, you would add the following section for the minimum properties required to establish a connection:

```
if (ds instanceof AutoRESTDataSource)
{
AutoRESTDataSource jmds = (AutoRESTDataSource) ds;
System.out.println("description=" + jmds.getDescription());
System.out.println("Config=" + jmds.getConfig());
System.out.println();
}

...
System.out.println();
}
```

Calling a data source in an application

Applications can call a Progress DataDirect data source using a logical name to retrieve the `javax.sql.DataSource` object. This object loads the specified driver and can be used to establish a connection to the database.

Once the data source has been registered with JNDI, it can be used by your JDBC application as shown in the following code example.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("EmployeeDB");
Connection con = ds.getConnection("domino", "spark");
```

In this example, the JNDI environment is first initialized. Next, the initial naming context is used to find the logical name of the data source (`EmployeeDB`). The `Context.lookup()` method returns a reference to a Java object, which is narrowed to a `javax.sql.DataSource` object. Then, the `DataSource.getConnection()` method is called to establish a connection.

Testing a data source connection

You can use DataDirect Test™ to establish and test a data source connection. The screen shots in this section were taken on a Windows system.

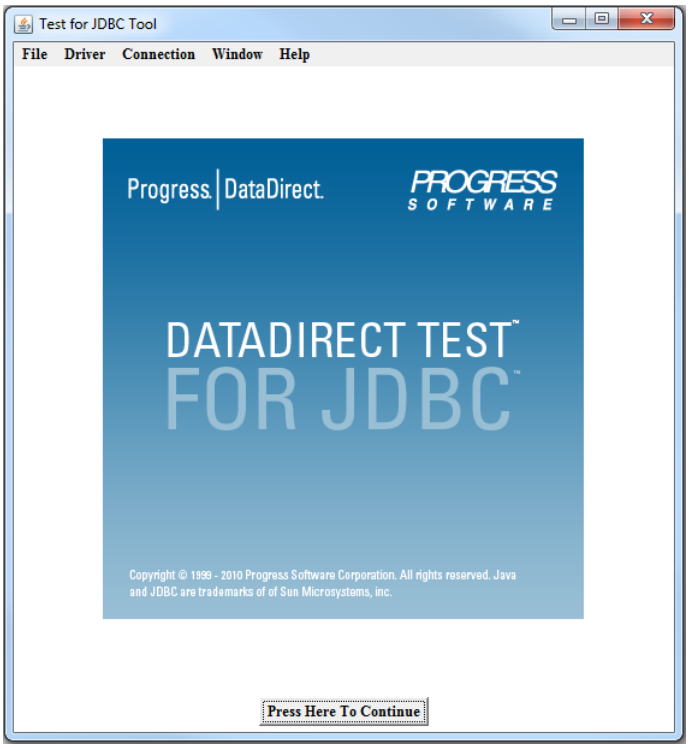
Take the following steps to establish a connection.

1. Navigate to the installation directory. The default location is:
   - Windows systems: `Program Files\Progress\DataDirect\JDBC\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC/testforjdbc`

   Note: For UNIX/Linux, if you do not have access to `/opt`, your home directory will be used in its place.

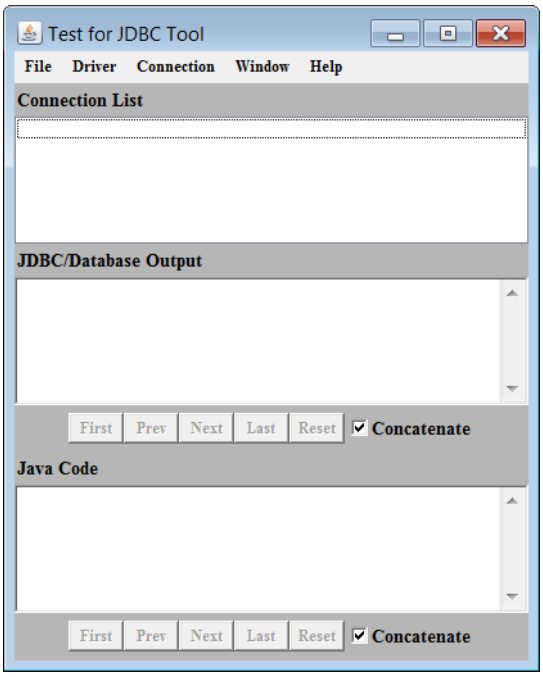
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The **Test for JDBC Tool** window appears:



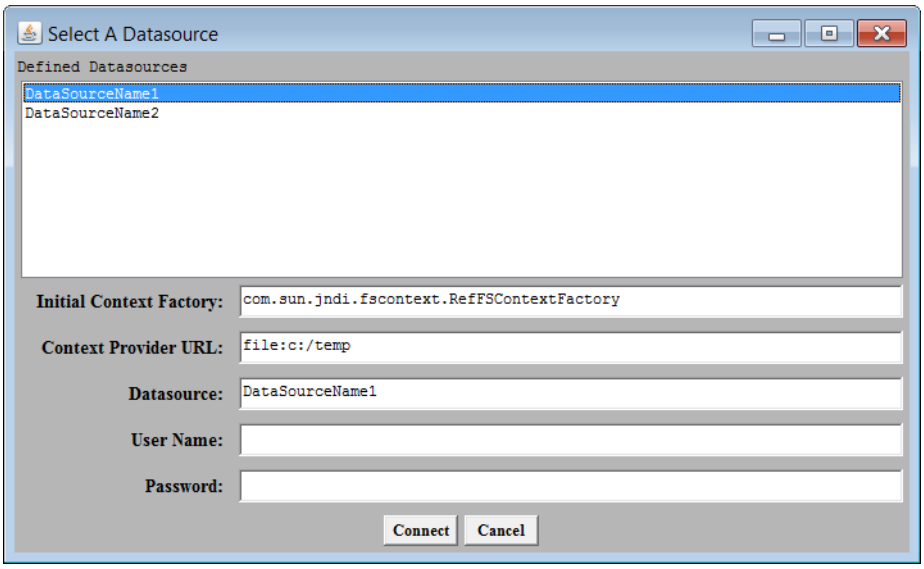
3. Click **Press Here to Continue**.

   The main dialog appears:



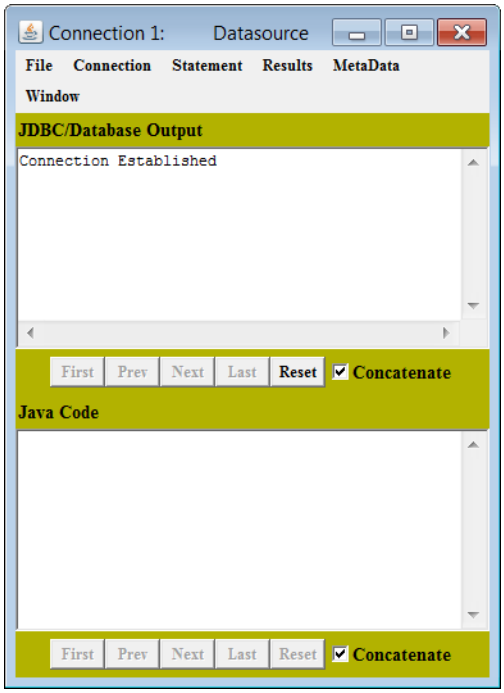
4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   The **Select A Database** dialog appears:



5. Select a datasource template from the **Defined Datasources** field.
6. Provide the following information:
   a) In the **Initial Context Factory**, specify the location of the initial context provider for your application.
   b) In the **Context Provider URL**, specify the location of the context provider for your application.
   c) In the **Datasource** field, specify the name of your datasource.
7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.
8. Click **Connect**.

If the connection information is entered correctly, the **JDBC/Database Output** window reports that a connection has been established. If a connection is not established, the window reports an error.



Authentication

The driver supports the following authentication methods:

- *No Authentication* is used for REST services that do not require authentication. This is often used for publicly available data, such as services for weather or earthquake data, governmental census or statistcal data, or internal lists of lookup codes.
- *Basic Authentication* authenticates using the specified user IDs, passwords, and HTTP headers.
- *AWS Authentication* authenticates using AWS (Amazon Web Services) credentials.
- *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. In some scenarios, the REST services may also authenticate the user ID.
- *URL Parameter Authentication* authenticates by passing security tokens using URLs. In some scenarios, the REST services may also authenticate the user ID.
- *OAuth 2.0 Authentication* authenticates 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=None`).

See also

AuthenticationMethod on page 125

Basic authentication

To configure the driver to use basic authentication:

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `Basic`.
- Set the AuthHeader property to specify the name of the HTTP header used for authentication. The default is `Authorization`.
- Set the User property to specify the user name that is used to connect to your REST service. For example, `jsmith`.
- Set the Password property to specify your password.
- Optionally, set the HealthURI property to specify the URI that the driver calls to confirm connectivity. Services using basic authentication do not perform an explicit action upon connection. You can work around this limitation by specifying a value for this property. 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 properties you want to configure.

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

For a connection URL:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=basic;
  Config=C:/path/to/myrest.rest;User=jsmith;Password=secret;");
```

For a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("Basic");
mds.setConfig("C:/path/to/myrest.rest");
```

See also

Connection property descriptions on page 111

AWS credentials authentication

To configure the driver to use AWS credentials authentication:

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `AWS`.
- Set the AccessKey property to specify your access key ID for your IAM user or AWS account root user.
- Set the Region property 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 SecretKey property to specify your secret access key for an IAM user or AWS account root user.
- Optionally, specify values for any additional properties you want to configure.

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

For a connection URL:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=AWS;
  Config=C:/path/to/myrest.rest;AccessKey=ABCDEFGHIJKL1EXAMPLE;Region=us-east-2;
  SecretKey=aBcdeFGhiJKLM/N1OPQRS/tUvWxyzYEXAMPLEKEY");
```

For a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("AWS");
mds.setConfig("C:/path/to/myrest.rest");
mds.setAccessKey("ABCDEFGHIJKL1EXAMPLE");
mds.setRegion("us-east-2");
mds.setSecretKey("aBcdeFGhiJKLM/N1OPQRS/tUvWxyzYEXAMPLEKEY");
```

See also

Connection property descriptions on page 111

Bearer token authentication

To configure the driver to use bearer token authentication:

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `BearerToken`.
- Set the SecurityToken property to specify your the API Token, configured as BearerToken, used for authentication.
- Optionally, specify values for any additional properties you want to configure.

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

For a connection URL:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=BearerToken;

Config=C:/path/to/myrest.rest;SecurityToken=C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx;");
```

For a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("BearerToken");
mds.setConfig("C:/path/to/myrest.rest");
mds.setBearerToken("C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx");
```

See also

Connection property descriptions on page 111

Digest authentication

To configure the driver to use digest authentication:

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `Digest`.
- Set the User property to specify the user name that is used to connect to your REST service. For example, `jsmith`.
- Set the Password property to specify your password.
- Optionally, set the HealthURI property to specify the URI that the driver calls to confirm connectivity. Services using digest authentication do not perform an explicit action upon connection. You can work around this limitation by specifying a value for this property. 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 properties you want to configure.

The following examples demonstrate a session using a Model file and digest authentication.

For a connection URL:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:autorest:https://example.com/;AuthenticationMethod=BearerToken;
  Config=C:/path/to/myrest.rest;User=jsmith;Password=secret;");
```

For a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("Digest");
mds.setConfig("C:/path/to/myrest.rest");
```

See also

Connection property descriptions on page 111

HTTP header authentication

To configure the driver to use HTTP header authentication:

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `HttpHeader`.
- Set the AuthHeader property to specify the name of the HTTP header used for authentication. The default is `Authorization`.
- Set the SecurityToken to specify the security token required to make a connection to your endpoint. For example, `XaBARTsLZReM`.
- Optionally, set the HealthURI property 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 property. 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 properties you want to configure.

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

For a connection URL:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:autorest:AuthenticationMethod=HttpHeader;
  Config=C:/path/to/myrest.rest;SecurityToken=XaBARTsLZReM;User=jsmith;");
```

For a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("HttpHeader");
mds.setConfig("C:/path/to/myrest.rest");
mds.setSecurityToken("XaBARTsLZReM");
mds.setUser("jsmith");
```

See also

Connection property descriptions on page 111

URL parameter authentication

To configure the driver to use HTTP header authentication:

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `UrlParameter`.
- Set the AuthParam property to specify the name of the URL parameter used to pass the security token. For example, `apikey`.
- Set the SecurityToken to specify the security token required to make a connection to your endpoint. For example, `XaBARTsLZReM`.
- If required by your service, set the User property to specify your logon ID.
- Optionally, set the HealthURI property to specify the URI that the driver calls to confirm connectivity. Services using URL parameter authentication do not perform an explicit action upon connection. You can work around this limitation by specifying a value for this property. 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 properties you want to configure.

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

For a connection URL:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:autorest:AuthenticationMethod=UrlParameter;AuthParam=apikey;
  Config=C:/path/to/myrest.rest;SecurityToken=XaBARTsLZReM;User=jsmith;");
```

For a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("UrlParameter");
mds.setAuthParam(apikey);
mds.setConfig("C:/path/to/myrest.rest");
mds.setSecurityToken("XaBARTsLZReM");
mds.setUser("jsmith");
```

See also

Connection property descriptions on page 111

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 properties. Note that your authentication flow may differ from the types listed here. If you are unsure of your requirements, contact your system administrator.

See also

Connection property descriptions on page 111

Access token flow

The access token authentication flow 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 99 for details.

To use an access token flow:

- The application should be configured to set the AccessToken property 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 properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/yelp.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `OAuth2-AccessToken`.

  Note: To support existing configurations, the AuthenticationMethod property will continue to support the `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 AuthHeader property to specify the name of the HTTP header used for authentication.
  - Set the SecurityToken property to specify the ID value of the HTTP header named by the AuthHeader option.
- Optionally, set the ClientCredentialsMode property to determine how client credentials must be specified in a request to obtain an access token when using OAuth 2.0. Configure this property for flows that require client credentials to be specified in only a basic authentication header or only as a URL parameter.
  - If set to `all`, the client credentials must be specified in a basic authentication header and as URL parameters. This is the default setting.
  - If set to `basic`, the client credentials must be specified using a basic authentication header.
  - If set to `url`, the client credentials must be specified as URL parameters.

The following examples demonstrate a basic configuration for Yelp™ using an access token flow:

Using a connection URL:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:autorest:AccessToken='C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx';
   AuthenticationMethod=OAuth2-AccessToken;Config=C:/path/to/yelp.rest;");
```

Using a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAccessToken("C3TQH9zjweek4CgJCU-4Mxb2DxPLNfI2LB3a-dNfpWnYx");
mds.setAuthenticationMethod("OAuth2-AccessToken");
mds.setConfig("C:/path/to/yelp.rest");
```

See also

Connection property descriptions on page 111

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 TokenURI property. If authentication takes place with a third-party authentication service, the application is redirected to the endpoint provided in the RedirectURI property to begin the session.

To use an authorization code grant:

- The application should be configured to set the OAuthCode property to specify the authorization code that is exchanged for the access token.
- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/box.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `OAuth2-AuthorizationCode`.

  Note: To support existing configurations, the AuthenticationMethod property will continue to support the `OAuth2` value for the authorization code grant.

- Set the ClientID property to specify the client ID key for your application.
- Set the TokenURI property 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 property. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- If required by your authentication flow, set the ClientSecret to specify client secret for your application.
- If required by your authentication flow, set the RedirectURI 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 AuthHeader property to specify the name of the HTTP header used for authentication.
  - Set the SecurityToken property to specify the value of the HTTP header named by the AuthHeader option.

  For example, if you have the header `Authorization:1a2bc34def567`, you would specify `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.

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

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

Using a connection URL:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-AuthorizationCode;
    ClientID='abcdefghik2lmn3o5qr67s';Config=C:/path/to/box.rest;
    OAuthCode='xyz123abc';TokenURI='https://api.box.com/oauth2/token';");
```

Using a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("OAuth2-AuthorizationCode");
mds.setClientID("abcdefghij1k2lmn3o4p5qr67s");
mds.setConfig("C:/path/to/box.rest");
mds.setOAuthCode("xyz123abc");
mds.setTokenURI("https://api.box.com/oauth2/token");
```

See also

Connection property descriptions on page 111

Client credentials grant

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

To configure the driver to use a client credentials grant:

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/googleanalytics.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `OAuth2-ClientCredentials`.

  Note: To support existing configurations, the AuthenticationMethod property will continue to support the `OAuth2` value for the client credentials grant.
- Set the ClientID property to specify the client ID key for your application.

- Set the ClientSecret property 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 TokenURI property 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 property. 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 AuthHeader property to specify the name of the HTTP header used for authentication.
  - Set the SecurityToken property to specify the value of the HTTP header named by the AuthHeader option.

  For example, if you have the header `Authorization:1a2bc34def567`, you would specify `AuthHeader=Authorization` and `SecurityToken=1a2bc34def567`.
- Optionally, set the Scope property to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the ClientCredentialsMode property to determine how client credentials are sent in a request in a request to obtain an access token. Configure this property for flows that require client credentials to be specified in only a basic authentication header or only as a URL parameter.
  - If set to `Default`, the client credentials are sent as a basic authentication header. This is the default setting.
  - If set to `Basic`, the client credentials are sent as a basic authentication header.
  - If set to `Url`, the client credentials are sent as a URL parameter.
  - If set to `Post`, the client credentials are sent in the body of a POST request.

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

Using a connection string:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-ClientCredentials;
  ClientID='123456789876-a1bc2de3fgh4ij567klmn8opqr9stuvw.apps.googleusercontent.com';
  ClientSecret='FaZBFRsGXTaR';Config=C:/path/to/googleanalytics.rest;
  TokenURI=https://accounts.google.com/o/oauth2/token;");
```

Using a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("OAuth2-ClientCredentials");
mds.setClientID("123456789876-a1bc2de3fgh4ij567klmn8opqr9stuvw.apps.googleusercontent.com");
mds.setClientSecret("FaZBFRsGXTaR");
mds.setConfig("C:/path/to/googleanalytics.rest");
mds.setTokenURI("https://accounts.google.com/o/oauth2/token");
```

See also

Connection property descriptions on page 111

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 AuthURI property. The driver then navigates to the endpoint specified by the TokenURI property to exchange the authorization code for the access and refresh tokens. Finally, the application is redirected to the location provided in the RedirectURI property 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 properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/box.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `OAuth2-AuthorizationCode`.

  Note: To support existing configurations, the AuthenticationMethod property will continue to support the `OAuth2` value for the authorization code grant.

- Set the ClientID property to specify the client ID key for your application.
- Set the ClientSecret property 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 AuthURI property to specify the endpoint for obtaining an authorization code.
- Set the TokenURI property 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 property. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- If required by your authentication flow, set the RedirectURI property 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 EnableLoginPrompt property to `true`. 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.
- Optionally, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
  - Set the AuthHeader property to specify the name of the HTTP header used for authentication.
  - Set the SecurityToken property to specify the value of the HTTP header named by the AuthHeader property.

  For example, if you have the header `Authorization:1a2bc34def567`, you would specify `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 property to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the ClientCredentialsMode property to determine how client credentials are sent in a request in a request to obtain an access token. Configure this property for flows that require client credentials to be specified in only a basic authentication header or only as a URL parameter.
  - If set to `Default`, the client credentials are sent as a basic authentication header. This is the default setting.
  - If set to `Basic`, the client credentials are sent as a basic authentication header.
  - If set to `Url`, the client credentials are sent as a URL parameter.
  - If set to `Post`, the client credentials are sent in the body of a POST request.

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

Using a connection URL:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-AuthorizationCode;
    AuthURI=https://api.box.com/oauth2/authorize;
    ClientID='abcdefghik2lmn3o5qr67s';ClientSecret=FaZBFRsGXTaR;
    Config=C:/path/to/box.rest;EnableLoginPrompt=1;
    RedirectURI=https://localhost:80;TokenURI='https://api.box.com/oauth2/token';");
```

Using a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("OAuth2-AuthorizationCode");
mds.setAuthURI("https://api.box.com/oauth2/authorize");
mds.setClientID("abcdefghij1k2lmn3o4p5qr67s");
mds.setClientSecret("FaZBFRsGXTaR");
mds.setConfig("C:/path/to/box.rest");
mds.setEnableLoginPrompt("1");
mds.setRedirectURI("https://localhost:80");
mds.setTokenURI("https://api.box.com/oauth2/token");
```

See also

Connection property descriptions on page 111

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 a JWT bearer grant flow:

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/docusign.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `OAuth2-JWTBearer`.

  Note: To support existing configurations, the AuthenticationMethod property will continue to support the `OAuth2` value for the JWT bearer grant.

- Set the ClaimsIssuer property to specify the client ID or consumer key of the authorization server.
- Set the ClaimsSubject property to specify your username.
- Set the JWTCertStore property 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 JWTCertPassword property to specify the password for the JWT certificate.
- Optionally, set the JWTCertAlias property to specify an alias for the JWT certificate.
- If required by your grant flow, set the TokenURI property to specify the endpoint used to exchange authentication credentials for access tokens.

- If required by your grant flow, set the RedirectURI to specify the endpoint that the client is returned to after authenticating with a third-party service.
- 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 AuthHeader property to specify the name of the HTTP header used for authentication.
  - Set the SecurityToken property to specify the value of the HTTP header named by the AuthHeader option.

  For example, if you have the header `Authorization:1a2bc34def567`, you would specify `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.

- If required by your grant flow, set the Scope property to 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 URL:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:autorest:Config=C:/path/to/docusign.rest;
      AuthenticationMethod=OAuth2-JWTBearer;ClaimsIssuer=1a2-b3c4-d5e6-f7g8-h9g;
      ClaimsSubject=1ab234cd-ef56-78gh;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 a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("OAuth2-JWTBearer");
mds.setAuthHeader("response_type");
mds.setClaimsIssuer("1ab234cd-ef56-78gh");
mds.setClaimsSubject("jsmith@example.com");
mds.setConfig("C:/path/to/docusign.rest");
mds.setJWTCertStore("jwtcert.jks");
mds.setRedirectUri("http://localhost:3000");
mds.setTokenUri("https://account-d.docusign.com/oauth/token");
mds.setScope("signature impersonation");
mds.setSecurityToken("code");
```

See also

Connection property descriptions on page 111

Password grant

The authentication flow for the password grant exchanges user credentials for the access token at the location specified by the TokenURI. 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 properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/zendesk.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `OAuth2-Password`.

  Note: To support existing configurations, the AuthenticationMethod property will continue to support the `OAuth2` value for the password grant.

- Set the User property to specify the user name that is used to fetch the access token from the Token endpoint.
- Set the Password property to specify the password used to fetch the access token.
- Set the TokenURI property 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 property. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- If required by your REST service, set the ClientID property to specify the client ID key for your application.
- If required by your REST service, set the ClientSecret property 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 AuthHeader property to specify the name of the HTTP header used for authentication.
  - Set the SecurityToken property to specify the value of the HTTP header named by the AuthHeader option.

  For example, if you have the header `Authorization:1a2bc34def567`, you would specify `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 property to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the ClientCredentialsMode property to determine how client credentials are sent in a request in a request to obtain an access token. Configure this property for flows that require client credentials to be specified in only a basic authentication header or only as a URL parameter.
  - If set to `Default`, the client credentials are sent as both a basic authentication header. This is the default setting.
  - If set to `Basic`, the client credentials are sent as a basic authentication header.
  - If set to `Url`, the client credentials are sent as a URL parameter.
  - If set to `Post`, the client credentials are sent in the body of a POST request.

The following example demonstrates a basic Zendesk™ session using a password grant:

Using a connection string:

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-Password;
   Config=C:/path/zendesk.rest;TokenURI=https://accounts.google.com/o/oauth2/token;
   User='jjones@example.com';Password='secretstuff';");
```

Using a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("OAuth2-Password");
mds.setConfig("C:/path/zendesk.rest");
mds.setTokenURI("https://accounts.google.com/o/oauth2/token");
mds.setUser("jjones@example.com");
mds.setPassword("secretstuff");
```

See also

Connection property descriptions on page 111

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 properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/box.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `OAuth2-PKCE`.

- Set the ClientID property to specify the client ID key for your application.
- Set the ClientSecret property 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 AuthURI property to specify the endpoint for obtaining an authorization code.
- Set the TokenURI property 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 property. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.

- Set the RedirectURI property 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`.
- Optionally, specify values for a custom HTTP header to be used for authentication, such as those used in tenant ID authentication:
  - Set the AuthHeader property to specify the name of the HTTP header used for authentication.
  - Set the SecurityToken property to specify the value of the HTTP header named by the AuthHeader property.

  For example, if you have the header `Authorization:1a2bc34def567`, you would specify `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 property to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the ClientCredentialsMode property to determine how client credentials are sent in a request in a request to obtain an access token. Configure this property for flows that require client credentials to be specified in only a basic authentication header or only as a URL parameter.
  - If set to `All`, the client credentials are sent as a basic authentication header. This is the default setting.
  - If set to `Basic`, the client credentials are sent as a basic authentication header.
  - If set to `Url`, the client credentials are sent as a URL parameter.
  - If set to `Post`, the client credentials are sent in the body of a POST request.

The following example demonstrates a simple configuration for a Spotify™ account using PKCE grant:

Using a connection URL:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-PKCE;
    AuthURI=https://accounts.spotify.com/authorize;
    ClientID='abcdefghik2lmn3o5qr67s';ClientSecret=FaZBFRsGXTaR;
    Config=C:/path/to/spotify.rest;RedirectURI=https://localhost:8080;
    TokenURI='https://accounts.spotify.com/api/token';");
```

Using a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("OAuth2-PKCE");
mds.setAuthURI("https://accounts.spotify.com/authorize");
mds.setClientID("abcdefghij1k2lmn3o4p5qr67s");
mds.setClientSecret("FaZBFRsGXTaR");
mds.setConfig("C:/path/to/spotify.rest");
mds.setRedirectURI("https://localhost:8080");
mds.setTokenURI("https://accounts.spotify.com/api/token");
```

See also

Connection property descriptions on page 111

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 TokenURI 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 a refresh token to support the access token flow. See Obtaining access and refresh tokens using the Configuration Manager on page 99 for details.

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

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/googleanalytics.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that you want to connect to and sample. For example, `https://example.com/countries/`.
- Set the AuthenticationMethod property to `OAuth2-RefreshToken`.

  Note: To support existing configurations, the AuthenticationMethod property will continue to support the `OAuth2` value for the refresh token grant.

- Set the ClientID property to specify the client ID key for your application.
- Set the ClientSecret property 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 RefreshToken property 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 TokenURI property 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 property. 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 AuthHeader property to specify the name of the HTTP header used for authentication.
  - Set the SecurityToken property to specify the value of the HTTP header named by the AuthHeader option.

  For example, if you have the header `Authorization:1a2bc34def567`, you would specify `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 property to specify a space-separated list of OAuth scopes to limit the permissions granted by the access token.
- Optionally, set the ClientCredentialsMode property to determine how client credentials are sent in a request in a request to obtain an access token. Configure this property for flows that require client credentials to be specified in only a basic authentication header or only as a URL parameter.
  - If set to `All`, the client credentials are sent as both a basic authentication header and a URL parameter. This is the default setting.
  - If set to `Basic`, the client credentials are sent as a basic authentication header.
  - If set to `Url`, the client credentials are sent as a URL parameter.
  - If set to `Post`, the client credentials are sent in the body of a POST request.

The following example demonstrates a basic Google Analytics™ session using a refresh token grant:

Using a connection URL:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:autorest:AuthenticationMethod=OAuth2-RefreshToken;
    ClientID='1234567898-a1bc2de3fgh4ij567klmn8opqr9stu.apps.googleusercontent.com'
    ClientSecret='FaZBFRsGXTaR';Config=C:/path/to/googleanalytics.rest;
    RefreshToken='1/abCD0F1GHijkLmNOPqrs_T2VWx3Y-Zabc45dE6FGh';
    TokenURI=https://accounts.google.com/o/oauth2/token;");
```

Using a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setAuthenticationMethod("OAuth2-RefreshToken");
mds.setClientID("1234567898-a1bc2de3fgh4ij567klmn8opqr9stu.apps.googleusercontent.com");
mds.setClientSecret("FaZBFRsGXTaR");
mds.setConfig("C:/path/to/googleanalytics.rest");
mds.setRefreshToken("1/abCD0F1GHijkLmNOPqrs_T2VWx3Y-Zabc45dE6FGh");
mds.setTokenURI("https://accounts.google.com/o/oauth2/token");
```

See also

Connection property descriptions on page 111

Obtaining access and refresh tokens using the Configuration Manager

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
- Scope: An OAuth scope, or a space-separated list of OAuth scopes, which specifies the permissions that limit application access to the 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 access and refresh tokens for either the access token flow or the refresh token grant. In addition, the Configuration Manager produces a connection URL that you can use in your application.

Note: You must allow popups in your browser to obtain access tokens with the Configuration Manager.

---

1. Open the Autonomous REST Configuration Manager by double-clicking the driver jar file. Or, in a command line, navigate to the directory containing your driver jar file; then, execute the following command:

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

   The Autonomous REST Configuration Manager opens in your default web browser.

2. Set **Authentication Method** to `OAuth2`.
3. Provide the following information in the fields provided.
   - **Sample**
   - **Authorization URI**
   - **Token URI**
   - **Client ID**

- **Client Secret**
- **Scope** (if required by your REST service)

4. Obtain access tokens.
   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 tokens. (This popup may not appear if you previously provided consent to the Configuration Manager.)
   d) The **Access Token** and **Refresh Token** fields populate with values 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 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 JDBC 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 into your JDBC 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 properties.

```
jdbc:datadirect:autorest:Sample=endpoint;AuthenticationMethod=OAuth2;
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.

```
jdbc:datadirect:autorest:Sample=endpoint;Authentication Method=OAuth2;
TokenURI=token_uri;ClientID=client_id;
ClientSecret=client_secret;AccessToken=access_token;
```

See also

Getting started using prebuilt Model files on page 18
Generating a Model file with the Autonomous REST Composer on page 59

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 Config property to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
- Set the AuthenticationMethod property to `Custom`. Note that `Custom` is the default when a custom authentication request is defined in the Model file.
- Set the CustomAuthParams property 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 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 ServerName property to set the host name portion of the HTTP endpoint to which you send requests.
- If required by your service, set the User property to specify your logon ID.
- If required by your service, set the Password property to specify your password.
- Optionally, specify values for any additional properties you want to configure.

The following examples demonstrate sessions using a Model file with custom authentication enabled:

For a connection URL:

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:autorest:Config=C:/path/to/myrest.rest;
  CustomAuthParams=123XYZ456abc789;My Company Inc;path/to/endpoint;
  ServerName=https://example.com;User=jsmith;Password=secret");
```

For a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setConfig("C:/path/to/myrest.rest");
mds.setCustomAuthParams("123XYZ456abc789;My Company Inc;path/to/endpoint");
mds.setServerName("https://example.com");
mds.setUser("jsmith");
mds.setPassword("secret");
```

See also

Custom authentication requests on page 188
Connection property descriptions on page 111

Data Encryption

TLS/SSL works by allowing the client and server to send each other encrypted data that only they can decrypt. TLS/SSL negotiates the terms of the encryption in a sequence of events known as the *handshake*. 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.

Configuring TLS/SSL Encryption

The driver supports TLS/SSL encryption for all supported REST services.

To configure SSL encryption:

Important: The driver complies with FIPS when FIPS mode is enabled with the client JVM. See "FIPS (Federal Information Processing Standard)" for more information.

- Configure the minimum properties required for a connection:
  - If you are using a Model file, set the Config property to provide the name and location of the Model file. For example, `C:/path/to/myrest.rest`.
  - If you are using the Sample property, set the Sample property to specify the endpoint that the want to connect to and sample. For example, `https://example.com/countries/`.
- Set the PortNumber property to specify the port number of the server listener. The default is `443`.
- Set the EncryptionMethod property to `SSL`.
- (Optional) Set the CryptoProtocolVersion property to specify acceptable cryptographic protocol versions (for example, TLSv1.2) supported by your server.
- (Optional) Specify the location and password of the truststore file used for SSL server authentication. Either set the TrustStore and TrustStorePassword properties or their corresponding Java system properties (`javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword`, respectively).
- (Optional) To validate certificates sent by the database server, set the ValidateServerCertificate property to `true`.
- (Optional) Set the HostNameInCertificate property to a host name to be used to validate the certificate. The HostNameInCertificate property 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.
- (Optional) If your database server is configured for SSL client authentication, configure your keystore information:
  - Specify the location and password of the keystore file. Either set the KeyStore and KeyStorePassword properties or their corresponding Java system properties (`javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword`, respectively).
  - If any key entry in the keystore file is password-protected, set the KeyPassword property to the key password.

The following examples demonstrate the required properties for a session using TLS/SSL encryption with no authentication.

For a connection URL:

```
Connection conn = DriverManager.getConnection
  ("jdbc:datadirect:autorest:https://example.com/;Config=C:/path/to/myrest.rest;
         EncryptionMethod=SSL");
```

For a data source:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
mds.setDescription("My Autonomous REST Data Source");
mds.setConfig ("C:/path/to/myrest.rest");
mds.setEncryptionMethod("SSL");
mds.setServerName("https://example.com/");
```

See also

Connection property descriptions on page 111
Configuring TLS/SSL Server Authentication on page 103
Configuring TLS/SSL Client Authentication on page 104

Configuring 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*. Optionally, the client may check the subject (owner) of the certificate. If the certificate matches a trusted CA in the truststore (and the certificate's subject matches the value that the application expects), an encrypted connection is established between the client and server. If the certificate does not match, the connection fails and the driver throws an exception.

To check the issuer of the certificate against the contents of the truststore, the driver must be able to locate the truststore and unlock the truststore with the appropriate password. You can specify truststore information in either of the following ways:

- Specify values for the Java system properties javax.net.ssl.trustStore and javax.net.ssl.trustStorePassword. For example:

  ```
  java -Djavax.net.ssl.trustStore=C:\Certificates\MyTruststore
       -Djavax.net.ssl.trustStorePassword=MyTruststorePassword
  ```

  This method sets values for all TLS/SSL sockets created in the JVM.

- Specify values for the connection properties TrustStore and TrustStorePassword in the connection URL. For example:

  `TrustStore=C:\Certficates\MyTruststore`

  and

  `TrustStorePassword=MyTruststorePassword`

  Any values specified by the TrustStore and TrustStorePassword properties override values specified by the Java system properties. This allows you to choose which truststore file you want to use for a particular connection.

Alternatively, you can configure the drivers 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. If the driver is configured to trust any certificate sent from the server, the issuer information in the certificate is ignored.

Configuring 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 has proved its identity. Similar to TLS/SSL 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*.

The driver must be able to locate the keystore and unlock the keystore with the appropriate keystore password. Depending on the type of keystore used, the driver also may need to unlock the keystore entry with a password to gain access to the certificate and its private key.

The drivers can use the following types of keystores:

- Java Keystore (JKS) contains a collection of certificates. Each entry is identified by an alias. The value of each entry is a certificate and the certificate's private key. Each keystore entry can have the same password as the keystore password or a different password. If a keystore entry has a password different than the keystore password, the driver must provide this password to unlock the entry and gain access to the certificate and its private key.
- PKCS #12 keystores. To gain access to the certificate and its private key, the driver must provide the keystore password. The file extension of the keystore must be .pfx or .p12.

You can specify this information in either of the following ways:

- Specify values for the Java system properties javax.net.ssl.keyStore and javax.net.ssl.keyStorePassword. For example:

  ```
  java -Djavax.net.ssl.keyStore=C:\Certificates\MyKeystore
       -Djavax.net.ssl.keyStorePassword=MyKeystorePassword
  ```

  This method sets values for all TLS/SSL sockets created in the JVM.

Note: If the keystore specified by the javax.net.ssl.keyStore Java system property is a JKS and the keystore entry has a password different than the keystore password, the KeyPassword connection property must specify the password of the keystore entry (for example, `KeyPassword=MyKeyPassword`).

- Specify values for the connection properties KeyStore and KeyStorePassword in the connection URL. For example:

  `KeyStore=C:\Certficates\MyKeyStore`

  and

  `KeyStorePassword=MyKeystorePassword`

Note: If the keystore specified by the KeyStore connection property is a JKS and the keystore entry has a password different than the keystore password, the KeyPassword connection property must specify the password of the keystore entry (for example, `KeyPassword=MyKeyPassword`).

Any values specified by the KeyStore and KeyStorePassword properties override values specified by the Java system properties. This allows you to choose which keystore file you want to use for a particular connection.

FIPS (Federal Information Processing Standard)

The Federal Information Processing Standard (or FIPS) is a cryptography standard created by the U.S. government. FIPS specifications require certain secure algorithms, cryptographic modules, and random number generation. The driver is FIPS compliant for data encryption when FIPS is enabled for the JVM on the client machine.

The following applies when the driver is running in a FIPS environment:

- The driver complies with 140-3 and 140-2 standards.
- The driver uses PKCS #11 providers to access keystores.

The driver was tested with FIPS 140-3 enabled using Red Hat OpenJDK 21 on a Red Hat Universal Base Image 9 instance.

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 8888, and provides a user ID and password for authentication:

```
java -Dhttp.proxyHost=pserver -Dhttp.proxyPort=8888 -Dhttp.proxyUser=smith
-Dhttp.proxyPassword=secret -cp autorest.jar com.acme.myapp.Main
```

Alternatively, you can use the ProxyHost, ProxyPort, ProxyUser, and ProxyPassword connection properties. See "Connection property descriptions" for details about these properties.

See also

Connection property descriptions on page 111

Performance considerations

EncryptionMethod: Data encryption may adversely affect performance because of the additional overhead (mainly CPU usage) required to encrypt and decrypt data.

InsensitiveResultSetBufferSize: To improve performance when using scroll-insensitive result sets, the driver can cache the result set data in memory instead of writing it to disk. By default, the driver caches 2 MB of insensitive result set data in memory and writes any remaining result set data to disk. Performance can be improved by increasing the amount of memory used by the driver before writing data to disk or by forcing the driver to never write insensitive result set data to disk. The maximum cache size setting is 2 GB.

FetchSize/**WSFetchSize**: The connection properties FetchSize and WSFetchSize can be used to adjust the trade-off between throughput and response time. In general, setting larger values for WSFetchSize and FetchSize 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 WSFetchSize is set to `500`, the driver must make 200 Web service calls to get the 100,000 rows. If, however, WSFetchSize is set to `2000` (the maximum), the driver only needs to make 50 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 FetchSize to 50 (the number of rows displayed on a page) and WSFetchSize 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.

ReadAhead: The ReadAhead property 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 property.
- 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 users, we strongly recommend specifying only smaller values for the ReadAhead property. 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.

WSPoolSize: WSPoolSize determines the maximum number of sessions the driver uses when there are multiple active connections. By increasing this number, you increase the number of sessions the driver uses to distribute calls to the REST service, thereby improving throughput and performance. For example, if WSPoolSize is set to 1, and you have two open connections, the session must complete a call from one connection before it can begin processing a call from the other connection. However, if WSPoolSize is equal to 2, a second session is opened that allows calls from both connections to be processed simultaneously.

Note: The number specified for WSPoolSize should not exceed the amount of sessions permitted by your REST service.

4

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
- Scrollable cursors
- Parameter metadata support
- ResultSet metaData support
- Rowset 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.

Scrollable cursors

The driver supports scroll-insensitive result sets and updatable result sets.

Note: When the driver cannot support the requested result set type or concurrency, it automatically downgrades the cursor and generates one or more SQLWarnings with detailed information.

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 = ?
```

ResultSet metaData support

If your application requires table name information, the driver can return table name information in ResultSet metadata for Select statements. The Select statements for which ResultSet metadata is returned may contain aliases, joins, and fully qualified names. The following queries are examples of Select statements for which the ResultSetMetaData.getTableName() method returns the correct table name for columns in the Select list:

```
SELECT id, name FROM Employee
SELECT E.id, E.name FROM Employee E
SELECT E.id, E.name AS EmployeeName FROM Employee E
SELECT E.id, E.name, I.location, I.phone FROM Employee E, EmployeeInfo I
   WHERE E.id = I.id
SELECT id, name, location, phone FROM Employee, EmployeeInfo WHERE id = empId
SELECT Employee.id, Employee.name, EmployeeInfo.location, EmployeeInfo.phone
   FROM Employee, EmployeeInfo WHERE Employee.id = EmployeeInfo.id
```

The table name returned by the driver for generated columns is an empty string. The following query is an example of a Select statement that returns a result set that contains a generated column (the column named "upper").

```
SELECT E.id, E.name as EmployeeName, {fn UCASE(E.name)} AS upper FROM Employee E
```

The driver also can return catalog name information when the ResultSetMetaData.getCatalogName() method is called if the driver can determine that information. For example, for the following statement, the driver returns "test" for the catalog name and "foo" for the table name:

```
SELECT * FROM test.foo
```

The additional processing required to return table name and catalog name information is only performed if the ResultSetMetaData.getTableName() or ResultSetMetaData.getCatalogName() methods are called.

Rowset support

The driver supports any JSR 114 implementation of the RowSet interface, including:

- CachedRowSets
- FilteredRowSets
- WebRowSets
- JoinRowSets
- JDBCRowSets

Visit https://www.jcp.org/en/jsr/detail?id=114 for more information about JSR 114.

5

Connection property descriptions

You can use connection properties to customize the driver for your environment. This section organizes connection properties according to functionality. You can use connection properties with either the JDBC `DriverManager` or a JDBC data source. For a `DriverManager` connection, a property is expressed as a key value pair and takes the form *`property=value`*. For a data source connection, a property is expressed as a JDBC method and takes the form `set`*`property`*`(`*`value`*`)`.

Note:

- In a JDBC data source, string values must be enclosed in double quotation marks, for example, `setUser("abc@defcorp.com")`.
- The data type listed for each connection property is the Java data type used for the property value in a JDBC data source.
- Connection property names are case-insensitive. For example, `Password` is the same as `password`.
- For connection properties that support string values, use the following escape sequence to specify values containing leading or trailing spaces and curly brackets: `{`*`value`*`}`. For example: `User={hello }` or `Password={{hello}}`.

The following tables describe the connection properties by functionality:

General properties

The following table summarizes connection properties which are that are used to connect to a REST service.

Table 12: General Properties

Property	Data Source Method	Default
Config on page 133	`getRestConfiguration()` `setRestConfiguration(URI)`	No default value
PortNumber on page 155	`getPortNumber()` `setPortNumber(Integer)`	No default value
Sample on page 164	`getSample()` `setSample(URI)`	No default value
ServerName on page 168	`getServerName()` `setServerName(String)`	No default value

Basic (user ID and password) authentication properties

The following table summarizes connection properties used for basic (user ID and password) authentication.

Table 13: Basic (user ID and password) authentication properties

Property	Data Source Method	Default
AuthenticationMethod on page 125	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`None`

Property	Data Source Method	Default
AuthHeader on page 127	`getAuthHeader()` `setAuthHeader(String)`	`Authorization`
AuthParam on page 127	`getAuthParam()` `setAuthParam(String)`	No default value
HealthURI on page 142	`getHealthUri()` `setHealthUri(URI)`	No default value
Password on page 154	`getPassword()` `setPassword(String)`	No default value
User on page 176	`getUser()` `setUser(String)`	No default value

AWS authentication properties

The following table summarizes connection properties used for AWS credentials authentication.

Table 14: AWS authentication properties

Property	Data Source Method	Default
AuthenticationMethod on page 125	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`None`
AccessKey on page 124	`getAccessKey()` `setAccessKey(String)`	No default value
Region on page 162	`getRegionName()` `setRegionName(String)`	No default value
SecretKey on page 166	`getSecretKey()` `setSecretKey(String)`	No default value

Bearer token authentication properties

The following table summarizes connection properties used for bearer token authentication.

Table 15: Bearer token authentication properties

Property	Data Source Method	Default
AuthenticationMethod on page 125	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`None`
HealthURI on page 142	`getHealthUri()` `setHealthUri(URI)`	No default value
SecurityToken on page 167	`getSecurityToken()` `setSecurityToken(String)`	No default value

Custom authentication properties

The following table summarizes connection properties used for custom authentication.

Table 16: Custom authentication properties

Property	Data Source Method	Default
AuthenticationMethod on page 125	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`None`
CustomAuthParams on page 135	`getCustomAuthParams()` `setCustomAuthParams(String)`	No default value
HealthURI on page 142	`getHealthUri()` `setHealthUri(URI)`	No default value
Password on page 154	`getPassword()` `setPassword(String)`	No default value
User on page 176	`getUser()` `setUser(String)`	No default value

Digest authentication properties

The following table summarizes connection properties used for digest authentication.

Table 17: Digest authentication properties

Property	Data Source Method	Default
AuthenticationMethod on page 125	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`None`

Property	Data Source Method	Default
HealthURI on page 142	`getHealthUri()` `setHealthUri(URI)`	No default value
Password on page 154	`getPassword()` `setPassword(String)`	No default value
User on page 176	`getUser()` `setUser(String)`	No default value

HTTP header authentication properties

The following table summarizes connection properties used for HTTP header authentication.

Table 18: HTTP header authentication properties

Property	Data Source Method	Default
AuthenticationMethod on page 125	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`None`
AuthHeader on page 127	`getAuthHeader()` `setAuthHeader(String)`	`Authorization`
HealthURI on page 142	`getHealthUri()` `setHealthUri(URI)`	No default value
SecurityToken on page 167	`getSecurityToken()` `setSecurityToken(String)`	No default value

OAuth 2.0 properties

The following table summarizes properties used for OAuth 2.0 authentication. The OAuth 2.0 properties 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 on page 84.

Table 19: OAuth 2.0 properties

Property	Data Source Method	Default
AccessToken on page 125	`getAccessToken()` `setAccessToken(String)`	No default value

Property	Data Source Method	Default
AuthenticationMethod on page 125	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`None`
AuthURI on page 128	`getAuthUri()` `setAuthUri(String)`	No default value
ClaimsIssuer on page 129	`getClaimsIssuer()` `setClaimsIssuer(String)`	No default value
ClaimsSubject on page 130	`getClaimsSubject()` `setClaimsSubject(String)`	No default value
ClientID on page 131	`getClientId()` `setClientId(String)`	No default value
ClientCredentialsMode on page 130	`getClientCredentialsMode()` `setClientCredentialsMode(String)`	`Default`
ClientSecret on page 132	`getClientSecret()` `setClientSecret(String)`	No default value
EnableLoginPrompt on page 138	`getEnableLoginPrompt()` `setEnableLoginPrompt(Boolean)`	`false`
HealthURI on page 142	`getHealthUri()` `setHealthUri(URI)`	No default value
JWTCertAlias on page 147	`getJwtCertAlias()` `setJwtCertAlias(String)`	No default value
JWTCertPassword on page 148	`getJwtCertPassword()` `setJwtCertPassword(String)`	No default value
JWTCertStore on page 148	`getJwtCertStore()` `setJwtCertStore(String)`	No default value

Property	Data Source Method	Default
LogoffURI on page 152	`getLogoffUri()` `setLogoffUri(String)`	No default value
OAuthCode on page 153	`getCode()` `setCode(String)`	No default value
RedirectURI on page 160	`getRedirUri()` `setRedirUri(String)`	No default value
RefreshToken on page 161	`getRefreshToken()` `setRefreshToken(String)`	No default value
Scope on page 166	`getScope()` `setScope(String)`	No default value
TokenURI on page 173	`getTokenUri()` `setTokenUri(String)`	No default value

URL parameter authentication properties

The following table summarizes connection properties used for user URL parameter authentication.

Table 20: URL parameter authentication properties

Property	Data Source Method	Default
AuthenticationMethod on page 125	`getAuthenticationMethod()` `setAuthenticationMethod(String)`	`None`
AuthParam on page 127	`getAuthParam()` `setAuthParam(String)`	No default value
HealthURI on page 142	`getHealthUri()` `setHealthUri(URI)`	No default value

Property	Data Source Method	Default
SecurityToken on page 167	`getSecurityToken()` `setSecurityToken(String)`	No default value
User on page 176	`getUser()` `setUser(String)`	No default value

Data encryption properties

The following table summarizes properties used for configuring data encryption.

Table 21: Data encryption properties

Property	Data Source Method	Default
CryptoProtocolVersion on page 134	`getCryptoProtocolVersion()` `setCryptoProtocolVersion(String)`	No default value
EncryptionMethod on page 139	`getEncryptionMethod()` `setEncryptionMethod(String)`	`NoEncryption`
HostNameInCertificate on page 143	`getHostNameInCertificate()` `setHostNameInCertificate(String)`	No default value
KeyPassword on page 149	`getKeyPassword()` `setKeyPassword(String)`	No default value
Keystore on page 150	`getKeystore()` `setKeystore(String)`	No default value
KeystorePassword on page 150	`getKeystorePassword()` `setKeystorePassword(String)`	No default value
Truststore on page 174	`getTruststore()` `setTruststore(String)`	No default value

Property	Data Source Method	Default
TruststorePassword on page 175	`getTruststorePassword()` `setTruststorePassword(String)`	No default value
ValidateServerCertificate on page 177	`getValidateServerCert()` `setValidateServerCert(Boolean)`	No default value

Proxy server properties

The following table summarizes proxy server connection properties.

Property	Data Source Method	Default
ProxyHost on page 155	`getProxyHost()` `setProxyHost(String)`	No default value
ProxyPassword on page 156	`getProxyPassword()` `setProxyPassword(String)`	No default value
ProxyPort on page 157	`getProxyPort()` `setProxyPort(Integer)`	`0` which means that the default value is determined by the ProxyHost property. For HTTP: `80` For HTTPS: `443`
ProxyUser on page 157	`getProxyUser()` `setProxyUser(String)`	No default value

Web service properties

The following table summarizes Web service connection properties.

Table 22: Timeout Properties

Property	Data Source Method	Default
StmtCallLimit on page 171	`getStmtCallLimit()` `setStmtCallLimit(Integer)`	`0` (no limit)
StmtCallLimitBehavior on page 172	`getStmtCallLimitBehavior()` `setStmtCallLimitBehavior(String)`	`ErrorAlways`

Property	Data Source Method	Default
WSFetchSize on page 177	`getWSFetchSize()` `setWSFetchSize(Integer)`	`2000`
WSPoolSize on page 178	`getWsPoolSize()` `setWsPoolSize(Integer)`	`1`
WSRetryCount on page 179	`getWSRetryCount()` `setWSRetryCount(Integer)`	`5`
WSTimeout on page 180	`getWSTimeout()` `setWSTimeout(Integer)`	`120`

Data type properties

The following table summarizes connection properties which care related to data type behavior.

Property	Data Source Method	Default
ConvertNull on page 134	`getConvertNull()` `setConvertNull(Boolean)`	`true`
JDBCBehavior on page 145	`getJdbcBehavior()` `setJdbcBehavior(Integer)`	`1` (data types described using JDBC 3.0-equivalent data types)

Statement pooling properties

The following table summarizes statement pooling connection properties.

Table 23: Statement Pooling Properties

Property	Data Source Method	Default
ImportStatementPool on page 143	`getImportStatementPool()` `setImportStatementPool(String)`	No default value
MaxPooledStatements on page 152	`getMaxPooledStatements()` `setMaxPooledStatements(Integer)`	`0`
RegisterStatementPoolMonitorMBean on page 163	`getRegisterStatementPoolMonitorMbean()` `setRegisterStatementPoolMonitorMbean(Boolean)`	`false`

Additional properties

The following table summarizes additional connection properties.

Table 24: Additional Properties

Property	Data Source Method	Default
DebugRecord on page 136	`getDebugRecord()` `setDebugRecord(String)`	No default value
DefaultQueryOptions on page 137	`getDefaultQueryOptions()` `setDefaultQueryOptions(String)`	No default value
FetchSize on page 140	`getFetchSize()` `setFetchSize(Integer)`	`100` (rows)
FileTransferPartSize on page 141	`getFileTransferPartSize()` `setFileTransferPartSize(Integer)`	No default value
InsensitiveResultSetBufferSize on page 144	`getInsensitiveResultsetBufferSize()` `setInsensitiveResultsetBufferSize(Integer)`	`2048`
LogConfigFile on page 151	`getLogConfigFile()` `setLogConfigFile(String)`	`ddlogging.properties`
QualifyNormalizedNames on page 158	`public String getQualifyNormalizedNames()` `public void setQualifyNormalizedNames(String)`	`No`
ReadAhead on page 159	`getReadAheadThreads()` `setReadAheadThreads(Integer)`	`0`
RefreshDirtyCache on page 161	`getRefreshDirtyCache()` `setRefreshDirtyCache(Boolean)`	`true`
SamplingFailureTolerance on page 165	`getSamplingFailureTolerance()` `setSamplingFailureTolerance(Integer)`	`-1`
SpyAttributes on page 168	`getSpyAttributes()` `setSpyAttributes(String)`	No default value

Property	Data Source Method	Default
Table on page 172	`getTable()` `setTable(String)`	No default value
TransactionMode on page 174	`getTransactionMode()` `setTransactionMode(String)`	`NoTransactions`

For details, see the following topics:

- AccessKey
- AccessToken
- AuthenticationMethod
- AuthHeader
- AuthParam
- AuthURI
- ClaimsIssuer
- ClaimsSubject
- ClientCredentialsMode
- ClientID
- ClientSecret
- Config
- ConvertNull
- CryptoProtocolVersion
- CustomAuthParams
- DebugRecord
- DefaultQueryOptions
- EnableLoginPrompt
- EncryptionMethod
- FetchSize
- FileTransferPartSize
- HealthURI
- HostNameInCertificate
- ImportStatementPool
- InsensitiveResultSetBufferSize

- JDBCBehavior
- JSONRoot
- JWTCertAlias
- JWTCertPassword
- JWTCertStore
- KeyPassword
- Keystore
- KeystorePassword
- LogConfigFile
- LogoffURI
- MaxPooledStatements
- OAuthCode
- Password
- PortNumber
- ProxyHost
- ProxyPassword
- ProxyPort
- ProxyUser
- QualifyNormalizedNames
- ReadAhead
- RedirectURI
- RefreshDirtyCache
- RefreshToken
- Region
- RegisterStatementPoolMonitorMBean
- Sample
- SamplingFailureTolerance
- Scope
- SecretKey
- SecurityToken
- ServerName
- SpyAttributes
- StmtCallLimit

AccessKey

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.

Data Source Methods

```
public String getAccessKey()
```

```
public void setAccessKey(String)
```

Default Value

No default value

Data Type

String

AccessToken

Purpose

Specifies the access token used to authenticate to REST endpoints with OAuth 2.0 enabled. Typically, this property 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 property 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 expire ten minutes after generation. Once connected, the access token remains valid till the session is disconnected.
- See "OAuth 2.0 authentication" for examples and more information.

Data Source Methods

`public String getAccessToken()`

`public void setAccessToken(String)`

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

AuthenticationMethod

Purpose

Determines which authentication method the driver uses during the course of a session.

Valid Values

`None` | `AWS` | `Basic` | `BearerToken` | `OAuth2` | `OAuth2-AccessToken` | `OAuth2-AuthorizationCode` | `OAuth2-PKCE` | `OAuth2-Password` | `OAuth2-ClientCredentials` | `OAuth2-RefreshToken` | `OAuth2-JWTBearer` | `HTTPHeader` | `URLParameter` | `Digest` | `Custom`

Behavior

If set to `None`, the driver does not attempt to authenticate.

If set to `AWS`, the driver uses AWS (Amazon Web Services) credentials for authentication. You must also configure the AccessKey, Region, and SecretKey properties.

If set to `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 properties, you must also configure the AuthHeader property if the name of your HTTP header is not `Authorization` (the default).

If set to `BearerToken`, the driver uses an API Token, configured as BearerToken, for authentication. The BearerToken is specified via the SecurityToken property.

If set to `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 properties you specify. This setting differs from other OAuth 2.0 values in that, when specified, it exposes all OAuth 2.0 related properties on the Configuration Manager, instead of only those related to a specified grant type. See "OAuth 2.0 authentication" for details.

If set to `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 `OAuth2-AuthorizationCode`, the driver uses authorization code grant to authenticate to REST endpoints. When `EnableLoginPrompt=true`, the driver uses dynamic authorization flow.

If set to `OAuth2-PKCE`, the driver uses the PKCE grant to authenticate to REST endpoints. See "OAuth 2.0 authentication" for details.

If set to `OAuth2-Password`, the driver uses the password grant authentication to authenticate to REST endpoints. See "OAuth 2.0 authentication" for details.

If set to `OAuth2-ClientCredentials`, the driver uses the client credentials grant to authenticate to REST endpoints. See "OAuth 2.0 authentication" for details.

If set to `OAuth2-RefreshToken`, the driver uses the refresh token grant to authenticate to REST endpoints. See "OAuth 2.0 authentication" for details.

If set to `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 `HttpHeader`, the driver passes security tokens via HTTP headers for authentication. You must also configure SecurityToken property and, if the name of your HTTP header is not `Authorization` (the default), the AuthHeader property.

If set to `URLParameter`, the driver passes security tokens via the URL for authentication. You must also configure the AuthParam and SecurityToken properties.

If set to `Digest`, the driver uses digest access authentication to negotiate username and password authentication. You must also configure the User and Password properties.

If set to `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.

Data Source Methods

```
public String getAuthenticationMethod()
```

```
public void setAuthenticationMethod(String)
```

Default Value

`None`

Data Type

String

See also

Authentication on page 78

AuthHeader

Purpose

Specifies the name of the HTTP header used for authentication. This property is used when Basic (`AuthenticationMethod=Basic`), Header-based token authentication (`AuthenticationMethod=HttpHeader`), or OAuth 2.0 authentication is enabled; otherwise, this property is ignored.

Valid Values

`auth_header`

where:

`auth_header`

is the name of the HTTP header used for authentication. For example, `X-Api-Key`.

Data Source Methods

```
public String getAuthHeader()
```

```
public void setAuthHeader(String)
```

Default Value

`Authorization`

Data Type

String

See also

Basic authentication on page 79
HTTP header authentication on page 83
OAuth 2.0 authentication on page 84

AuthParam

Purpose

Specifies the name of the URL parameter used to pass the security token. This property is required when using URL parameters to pass tokens for authentication (AuthenticationMethod=UrlParameter); otherwise, this property 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`.

Data Source Methods

```
public String getAuthParam()
```

```
public void setAuthParam(String)
```

Default Value

No default value

Data Type

String

See also

URL parameter authentication on page 84

AuthURI

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.

Data Source Methods

```
public String getAuthUri()
```

```
public void setAuthUri(String)
```

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

ClaimsIssuer

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 property 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.

Data Source Methods

```
public String getClaimsIssuer()
```

```
public void setClaimsIssuer(String)
```

Default Value

No default value

Data Type

String

ClaimsSubject

Purpose

Specifies the identifier of the principle that is the subject of the JWT. This property 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.

Data Source Methods

```
public String getClaimsSubject()
```

```
public void setClaimsSubject(String)
```

Default Value

No default value

Data Type

String

ClientCredentialsMode

Purpose

Determines how client credentials are sent in a request to obtain an access token when using OAuth 2.0. Configure this property for flows that require client credentials to be specified as only a basic authentication header or as only a URL parameter.

Valid Values

`Default` | `Basic` | `URL` | `Post`

Behavior

If set to `Default`, the client credentials are sent as a basic authentication header.

If set to `Basic`, the client credentials are sent as a basic authentication header.

If set to `Url`, the client credentials are sent as a URL parameter.

If set to `Post`, the client credentials are sent in the body of a POST request.

Notes

- This property is not required for all authentication flows. If you are unsure of the requirements for your authentication flow, contact your administrator for more information.

Data Source Methods

```
public String getClientCredentialsMode()
```

```
public void setClientCredentialsMode(String)
```

Default Value

`Default`

Data Type

String

ClientID

Purpose

Specifies the client ID key for your application when authenticating to REST endpoints with OAuth 2.0 enabled.

Valid Values

`String`

where:

`String`

is the client ID key for your application.

Notes

- In some cases, the value for this property 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.

Data Source Methods

```
public String getClientId()
```

```
public void setClientId(String)
```

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

ClientSecret

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.

Data Source Methods

```
public String getClientSecret()
```

```
public void setClientSecret(String)
```

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

Config

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

For more information, see "Creating a Model file."

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 Config property, for specifying a Model File, or the Sample property, for specifying a single endpoint to sample. However, note that these properties are not required for all use cases, such as when the driver is being used to execute dynamically created stored procedures or functions.

Data Source Methods

```
public URI getConfig()
```

```
public void setConfig(URI)
```

Default Value

No default value

Data Type

URI

See also

Generating a Model file with the Autonomous REST Composer on page 59
Sample on page 164

ConvertNull

Purpose

Controls how data conversions are handled for null values.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver checks the data type being requested against the data type of the table column that stores the data. If a conversion between the requested type and column type is not defined, the driver generates an "unsupported data conversion" exception regardless of whether the column value is `NULL`.

If set to `false`, the driver does not perform the data type check if the value of the column is `NULL`. This allows null values to be returned even though a conversion between the requested type and the column type is undefined.

Data Source Methods

```
public Boolean getConvertNull()
```

```
public void setConvertNull(Boolean)
```

Default Value

`true`

Data Type

Boolean

CryptoProtocolVersion

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`

Data Source Methods

```
public String getCryptoProtocolVersion()
```

```
public void setCryptoProtocolVersion(String)
```

Default Value

No default value

Data Type

String

See also

Data Encryption on page 102

CustomAuthParams

Purpose

Specifies a list of parameter values used by custom authentication requests that are defined in the Model file. This property 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 property 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 property.

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 property, 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 property.

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 property:

```
CustomAuthParams=123XYZ456abc789;My Company Inc;www.example.com
```

Notes

- This property is enabled when `AuthenticationMethod=Custom`; otherwise, it is ignored.
- The values specified for this property are case insensitive.

Data Source Methods

```
public String getCustomAuthParams()
```

```
public void setCustomAuthParams(String)
```

Default Value

No default value

Data Type

String

See also

Custom authentication on page 100
Custom authentication requests on page 188

DebugRecord

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`.

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 JDBC Drivers Reference*.
- For assistance, contact Technical Support.

Data Source Methods

```
public String getDebugRecord()
```

```
public void setDebugRecord(String)
```

Default Value

No default value

Data Type

String

See also

Contacting Technical Support on page 52

DefaultQueryOptions

Purpose

Specifies a semicolon-separated list of parameters that are used as default filter values (Where clauses) for SQL queries.

Valid Values

string

where:

string

is a set of parameters and default filter values that you want to apply to SQL queries.

Notes

- A Where clause used in a SQL query overrides the DefaultQueryOptions passed in a connection URL.

Data Source Methods

```
public String getDefaultQueryOptions()
```

```
public void setDefaultQueryOptions(String)
```

Default Value

No default value

Data Type

String

EnableLoginPrompt

Purpose

Specifies whether the driver fetches access and refresh tokens at connection when OAuth 2.0 Authorization Code Grant is enabled (`OAuth2-AuthorizationCode`). When this property 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 property provides a method of fetching access and refresh tokens without using the Configuration Manager or third-party tool.

Valid Values

`true` | `false`

Behavior

If set to `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 `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 property is used only for the dynamic authorization code grant flow. See "Dynamic authorization code grant" for a full list of requirements.
- When this property is enabled, the value of the `RedirectURI` property must include the port number. For example, `RedirectURI=http://localhost:80` or `RedirectURI=http://localhost:8080`.

Data Source Methods

```
public Boolean getEnableLoginPrompt()
```

```
public void setEnableLoginPrompt(Boolean)
```

Default Value

`false`

Data Type

Boolean

EncryptionMethod

Purpose

Determines whether data is encrypted and decrypted when transmitted over the network between the driver and REST service.

Valid Values

`NoEncryption` | `SSL`

Behavior

If set to `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 `noEncryption`, data is not encrypted or decrypted.

Data Source Methods

```
public String getEncryptionMethod()
```

```
public void setEncryptionMethod(String)
```

Default Value

- SSL encryption is enabled when the URL specified in the Sample property or Model file uses HTTPS, regardless of the setting of EncryptionMethod.
- When SSL is enabled, the following properties also apply:
  - CryptoProtocolVersion
  - HostNameInCertificate
  - KeyPassword (for SSL client authentication)
  - KeyStore (for SSL client authentication)
  - KeyStorePassword (for SSL client authentication)
  - TrustStore
  - TrustStorePassword
  - ValidateServerCertificate

Data Type

String

See also

Data Encryption on page 102
Performance considerations on page 105

FetchSize

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 FetchSize 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 FetchSize when the result set contains exceptionally wide rows. Alternatively, the driver processes the number of rows specified by FetchSize when the result set contains rows of unexceptional width.
- FetchSize and WSFetchSize 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.
- You can use FetchSize to reduce demands on memory and decrease the likelihood of out-of-memory errors. Simply, decrease FetchSize to reduce the number of rows the driver is required to process before returning data to the application.

Data Source Methods

```
public Integer getFetchSize()
```

```
public void setFetchSize(Integer)
```

Default Value

`100`

Data Type

Integer

See also

Performance considerations on page 105

FileTransferPartSize

Purpose

Specifies the maximum number of bytes the driver can use for data transfer during upload or download operations. This setting serves as a guideline for the driver's internal buffer size. The driver may adjust the actual number of bytes—either increasing or decreasing—to comply with client operating system constraints or server feature requirements.

Valid Values

`0` | *x*

where:

x

is a positive integer representing the number of bytes to use, up to a maximum of 2147483647.

Behavior

If set to `0`, the driver uses 100MB (=104857600 bytes) as the default part size.

If set to *x*, the driver limits the number of bytes for each transfer in upload or download operations.

Notes

- This setting does not represent the maximum file size; the file can be smaller or larger than the specified value.
- If the specified value exceeds the file size, the driver uses the actual file size for transfer.
- If the specified value is smaller than the file size, the driver performs multiple iterations to transfer the entire file.
- If a value is not specified, the driver uses the default part size.

Data Source Methods

```
public Integer getFileTransferPartSize()
```

```
public void setFileTransferPartSize(Integer)
```

Default Value

No default value

Data Type

Integer

HealthURI

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 property. 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 property, executing a test connect on the Configuration Manager will return a message that the connection was successful every time.

Data Source Methods

```
public URI getHealthUri()
```

```
public void setHealthUri(URI)
```

Default Value

No default value

Data Type

URI

HostNameInCertificate

Purpose

Specifies a host name for certificate validation when SSL encryption is enabled and validation is enabled (`ValidateServerCertificate=true`). This property 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 property is ignored.
- If SSL encryption and validation is enabled and this property is unspecified, the driver uses the server name that is specified in the connection URL or data source of the connection to validate the certificate.

Data Source Methods

```
public String getHostNameInCertificate()
```

```
public void setHostNameInCertificate(String)
```

Default Value

No default value

Data Type

String

ImportStatementPool

Purpose

Specifies the path and file name of the file to be used to load the contents of the statement pool. When this property is specified, statements are imported into the statement pool from the specified file.

Valid Values

String

where:

String

is the path and file name of the file to be used to load the contents of the statement pool.

Notes

- If the driver cannot locate the specified file when establishing the connection, the connection fails and the driver throws an exception.
- For more information, refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public String getImportStatementPool()
```

```
public void setImportStatementPool(String)
```

Default Value

No default value

Data Type

String

InsensitiveResultSetBufferSize

Purpose

Determines the amount of memory that is used by the driver to cache insensitive result set data.

Valid Values

`-1` | `0` | *x*

where:

x

is a positive integer that represents the amount of memory.

Behavior

If set to `-1`, the driver caches insensitive result set data in memory. If the size of the result set exceeds available memory, an OutOfMemoryException is generated. With no need to write result set data to disk, the driver processes the data efficiently.

If set to `0`, the driver caches insensitive result set data in memory, up to a maximum of 2 MB. If the size of the result set data exceeds available memory, then the driver pages the result set data to disk, which can have a negative performance effect. Because result set data may be written to disk, the driver may have to reformat the data to write it correctly to disk.

If set to *x*, the driver caches insensitive result set data in memory and uses this value to set the size (in KB) of the memory buffer for caching insensitive result set data. If the size of the result set data exceeds available memory, then the driver pages the result set data to disk, which can have a negative performance effect. Because the result set data may be written to disk, the driver may have to reformat the data to write it correctly to disk. Specifying a buffer size that is a power of 2 results in efficient memory use.

Data Source Methods

```
public Integer getInsensitiveResultsetBufferSize()
```

```
public void setInsensitiveResultsetBufferSize(Integer)
```

Default Value

2048

Data Type

Integer

JDBCBehavior

Purpose

Determines how the driver describes database data types that map to the following JDBC 4.0 data types: NCHAR, NVARCHAR, NLONGVARCHAR, NCLOB, and SQLXML.

Valid Values

0 | 1

Behavior

If set to 0, the driver describes the data types as JDBC 4.0 data types.

If set to 1, the driver describes the data types using JDBC 3.0-equivalent data types regardless of the JVM. This allows your application to continue using JDBC 3.0 types.

Data Source Methods

```
public Integer getJdbcBehavior()
```

```
public void setJdbcBehavior(Integer)
```

Default Value

1

Data Type

Integer

JSONRoot

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 Sample property method, this property 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 property, 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 property 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 property is used only when specifying an endpoint using the Sample property 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 property 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 property.

Data Source Methods

```
public String getJSONRoot()
```

```
public void setJSONRoot(String)
```

Default Value

No default value

Data Type

String

JWTCertAlias

Purpose

Specifies the alias for the JWT certificate stored in the keystore when authenticating using the OAuth 2.0 JWT bearer grant type. This property is optional.

Valid Values

String

where:

String

is the alias for the JWT certificate.

Notes

- See "OAuth 2.0 authentication" for more information.

Data Source Methods

```
public String getJwtCertAlias()
```

```
public void setJwtCertAlias(String)
```

Default Value

No default value

Data Type

String

JWTCertPassword

Purpose

Specifies the password of the JWT certificate when authenticating using the OAuth 2.0 JWT bearer grant type. A value for this property 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.

Data Source Methods

```
public String getJwtCertPassword()
```

```
public void setJwtCertPassword(String)
```

Default Value

No default value

Data Type

String

JWTCertStore

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 property 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.

Data Source Methods

```
public String getJwtCertStore()
```

```
public void setJwtCertStore(String)
```

Default Value

No default value

Data Type

String

KeyPassword

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.

Data Source Methods

```
public String getKeyPassword()
```

```
public void setKeyPassword(String)
```

Default Value

No default value

Data Type

String

See also

Data Encryption on page 102

Keystore

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.

Data Source Methods

```
public String getKeystore()
```

```
public void setKeystore(String)
```

Default Value

No default value

Data Type

String

KeystorePassword

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.

Data Source Methods

```
public String getKeystorePassword()
```

```
public void setKeystorePassword(String)
```

Default Value

No default value

Data Type

String

See also

Data Encryption on page 102

LogConfigFile

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 "Using Java Logging" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public String getLogConfigFile()
```

```
public void setLogConfigFile(String)
```

Default Value

`ddlogging.properties`

Data Type

String

LogoffURI

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 (`AuthenticationMethod=OAuth2`).

Valid Values

`string`

where:

`string`

is the endpoint used to retrieve OAuth 2.0 authorization codes. For example: `https://example.com/oauth2/logout/`

Data Source Methods

`public String getLogoffUri()`

`public void setLogoffUri(String)`

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

MaxPooledStatements

Purpose

Specifies the maximum number of prepared statements to be pooled for each connection and enables the driver's internal prepared statement pooling when set to an integer greater than zero (0). The driver's internal prepared statement pooling provides performance benefits when the driver is not running from within an application server or another application that provides its own statement pooling.

Valid Values

`0` | *`x`*

where:

x

is a positive integer that represents a number of prepared statements to be cached.

Behavior

If set to `0`, the driver's internal prepared statement pooling is not enabled.

If set to *x*, the driver's internal prepared statement pooling is enabled and the driver uses the specified value to cache up to that many prepared statements created by an application. If the value set for this property is greater than the number of prepared statements that are used by the application, all prepared statements that are created by the application are cached. Because CallableStatement is a sub-class of PreparedStatement, CallableStatements also are cached.

Example

If the value of this property is set to `20`, the driver caches the last 20 prepared statements that are created by the application.

Notes

When you enable statement pooling, your applications can access the Statement Pool Monitor directly with DataDirect-specific methods. However, you can also enable the Statement Pool Monitor as a JMX MBean. To enable the Statement Pool Monitor as an MBean, statement pooling must be enabled with MaxPooledStatements and the Statement Pool Monitor MBean must be registered using the RegisterStatementPoolMonitorMBean connection property.

Data Source Methods

```
public Integer getMaxPooledStatements()
```

```
public void setMaxPooledStatements(Integer)
```

Default Value

`0`

Data Type

Integer

OAuthCode

Purpose

Specifies the temporary authorization code that is exchanged for access tokens when OAuth 2.0 authentication is enabled.

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 TokenURI property. If authentication is successful, an access token is generated and fetched from the specified location. Typically, this property is configured by the application.
- See "OAuth 2.0 authentication" for more information.

Data Source Methods

```
public String getCode()
```

```
public void setCode(String)
```

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

Password

Purpose

A password that is used to connect to the service.

Important: Setting the password using a data source is not recommended. The data source persists all properties, including password, in clear text.

Behavior

password

where:

password

is a valid password. The password is case-sensitive.

Data Source Methods

```
public String getPassword()
```

```
public void setPassword(String)
```

Default Value

No default value

Data Type

String

See also

Authentication on page 78

PortNumber

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.

Data Source Methods

```
public Integer getPortNumber()
public void setPortNumber(Integer)
```

Default Value

When SSL encryption is disabled:

80

When SSL encryption is enabled:

443

Data Type

Integer

ProxyHost

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.

Data Source Methods

```
public String getProxyHost()
public void setProxyHost(String)
```

Default Value

No default value

Data Type

String

See also

Connecting through a proxy server on page 105

ProxyPassword

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.

Data Source Methods

```
public String getProxyPassword()
public void setProxyPassword(String)
```

Default Value

No default value

Data Type

String

See also

Connecting through a proxy server on page 105

ProxyPort

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.

Data Source Methods

```
public Integer getProxyPort()
```

```
public void setProxyPort(Integer)
```

Default Value

`0` which means that the default value is determined by whether the value specified for the ProxyHost property is an HTTP or HTTPS URL.

For HTTP: `80`

For HTTPS: `443`

Data Type

Integer

See also

Connecting through a proxy server on page 105

ProxyUser

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.

Data Source Methods

```
public String getProxyUser()
```

```
public void setProxyUser(String)
```

Default Value

No default value

Data Type

String

See also

Connecting through a proxy server on page 105

QualifyNormalizedNames

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

`No` | `Table` | `FullPath`

Behavior

If set to `No`, the relational table name is derived solely from the column name of the array.

If set to `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 `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 property also controls the name of the foreign key column in the child table. For example, if this property 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.

Data Source Methods

```
public String getQualifyNormalizedNames()
```

```
public void setQualifyNormalizedNames(String)
```

Default Value

No

Data Type

String

See also

Mapping objects to tables on page 31

ReadAhead

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 property 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 property. 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 property 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 property.
  - 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.

Data Source Methods

```
public Integer getReadAheadThreads()
```

```
public void setReadAheadThreads(Integer)
```

Default Value

`0`

Data Type

Integer

See also

Performance considerations on page 105

RedirectURI

Purpose

Specifies the endpoint to which the client is returned after third-party authorization for OAuth 2.0 implementations.

Valid Values

`String`

where:

`String`

is the endpoint to which the client is returned after third-party authorization. For example, `http://localhost`.

Notes

- The redirect URI 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.

Data Source Methods

```
public String getRedirUri()
```

```
public void setRedirUri(String)
```

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

RefreshDirtyCache

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

`true | false`

Behavior

If set to `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 `false`, a dirty cache is not refreshed when the cache is referenced in a fetch operation.

Data Source Methods

```
public Boolean getRefreshDirtyCache()
```

```
public void setRefreshDirtyCache(Boolean)
```

Default Value

`true`

Data Type

Boolean

RefreshToken

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 REST endpoints service.

Notes

- See "OAuth 2.0 authentication" for more information.

Data Source Methods

```
public String getRefreshToken()
```

```
public void setRefreshToken(String)
```

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

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`.

Data Source Methods

```
public String getRegionName()
```

```
public void setRegionName(String)
```

Default Value

No default value

Data Type

String

RegisterStatementPoolMonitorMBean

Purpose

Registers the Statement Pool Monitor as a JMX MBean when statement pooling has been enabled with MaxPooledStatements. This allows you to manage statement pooling with standard JMX API calls and to use JMX-compliant tools, such as JConsole.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver registers an MBean for the statement pool monitor for each statement pool. This gives applications access to the Statement Pool Monitor through JMX when statement pooling is enabled.

If set to `false`, the driver does not register an MBean for the Statement Pool Monitor for any statement pool.

Notes

- Registering the MBean exports a reference to the Statement Pool Monitor. The exported reference can prevent garbage collection on connections if the connections are not properly closed. When garbage collection does not take place on these connections, out of memory errors can occur.
- For more information, refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public Boolean getRegisterStatementPoolMonitorMbean()
```

```
public void setRegisterStatementPoolMonitorMbean(Boolean)
```

Default Value

`false`

Data Type

Boolean

See also

MaxPooledStatements on page 152

Sample

Purpose

Specifies the endpoint that the driver connects to and samples. This property 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 property, you must create a Model file and specify its location with the Config property:

- 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

- The Table connection property allows you to determine the name of the table the specified endpoint maps to. If you do not provide a table name, the driver will determine the name based on the name of the endpoint.
- When using the Sample 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.
- In most scenarios, you will configure the Config property, for specifying a Model File, or the Sample property, for specifying a single endpoint to sample. However, note that these properties are not required for all use cases, such as when the driver is being used to execute dynamically created stored procedures or functions.

Data Source Methods

```
public URI getSample()
```

```
public void setSample(URI)
```

Default Value

No default value

Data Type

URI

See also

Config on page 133
Table on page 172

SamplingFailureTolerance

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.

Data Source Methods

```
public Integer getSamplingFailureTolerance()
```

```
public void setSamplingFailureTolerance(Integer)
```

Default Value

`-1`

Data Type

Integer

See also

Reviewing the status of your endpoints on page 69

Scope

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.

Data Source Methods

`public String getScope()`

`public void setScope(String)`

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

SecretKey

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.

Data Source Methods

`public String getSecretKey()`

```
public void setSecretKey(String)
```

Default Value

No default value

Data Type

String

SecurityToken

Purpose

Specifies the security token or other HTTP header value used to make a connection to the server.

This property is required when token-based authentication is enabled (`AuthenticationMethod=BearerToken | HttpHeader | 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 property specifies the value of custom HTTP headers named by the AuthHeader property. If a value for the AuthHeader property is specified, and you do not supply a value for this property, 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 a DataSource class, be aware that the SecurityToken property, like all DataSource properties, 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.

Data Source Methods

```
public String getSecurityToken()
```

```
public void setSecurityToken(String)
```

Default Value

No default value

Data Type

String

See also

Authentication on page 78

ServerName

Purpose

Specifies the host name portion of the HTTP endpoint to which you send requests. This property allows you to define endpoints without storing the host name component in the Model file.

Valid Values

url | *ip_address*

where:

url

is the root of the endpoint to which you send requests. For example, `https://example.com`.

ip_address

is the IP address of the endpoint to which you send requests.

Notes

- Specifying an HTTPS endpoint using the ServerName property enables SSL data encryption.
- The HostName property is an alias for the Server Name (ServerName) property.

Data Source Methods

```
public String getServerName()
public void setServerName(String)
```

Default Value

No default value

Data Type

String

SpyAttributes

Purpose

Enables DataDirect Spy to log detailed information about calls that are issued by the driver on behalf of the application. DataDirect Spy is not enabled by default.

Valid Values

(*spy_attribute*[;*spy_attribute*]...)

where:

spy_attribute

is any valid DataDirect spy attribute.

Behavior

Attribute	Description
`linelimit=`*`numberofchars`*	Sets the maximum number of characters that DataDirect Spy logs on a single line. The default is 0 (no maximum limit).
`log=(file)`*`filename`*	Directs logging to the file specified by *`filename`*. For Windows, if coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes`

`log=(filePrefix)`*`file_prefix`*	Directs logging to a file prefixed by *`file_prefix`*. The log file is named *`file_prefixX`*`.log`
	where:
	X is an integer that increments by 1 for each connection on which the prefix is specified.
	For example, if the attribute `log=(filePrefix)C:\\temp\\spy_` is specified on multiple connections, the following logs are created:
	`C:\temp\spy_1.log`
	`C:\temp\spy_2.log`
	`C:\temp\spy_3.log`
	`...`
	If coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash.
	For example:
	`log=(filePrefix)C:\\temp\\spy_;logIS=yes;logTName=yes`

`log=System.out`	Directs logging to the Java output standard, `System.out`.

Attribute	Description
logIS={yes | no | nosingleread}	Specifies whether DataDirect Spy logs activity on InputStream and Reader objects. When logIS=nosingleread, logging on InputStream and Reader objects is active; however, logging of the single-byte read InputStream.read or single-character Reader.read is suppressed to prevent generating large log files that contain single-byte or single character read messages. The default is no.
logLobs={yes | no}	Specifies whether DataDirect Spy logs activity on BLOB and CLOB objects.
logTName={yes | no}	Specifies whether DataDirect Spy logs the name of the current thread. The default is no.
timestamp={yes | no}	Specifies whether a timestamp is included on each line of the DataDirect Spy log. The default is no.

Example

The following value instructs the driver to log all JDBC activity to a file using a maximum of 80 characters for each line.

```
(log=(file)/tmp/spy.log;linelimit=80)
```

Notes

- If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: log=(file)C:\\temp\\spy.log.
- If a log file name does not include the .log extension, the driver automatically appends it. For example, a file named spy.jsp is renamed to spy.jsp.log by the driver.
- For more information, refer to "Tracking JDBC calls with DataDirect Spy" in the *Progress DataDirect for JDBC Drivers Reference*.

Data Source Methods

```
public String getSpyAttributes()
```

```
public void setSpyAttributes(String)
```

Default Value

No default value

Data Type

String

StmtCallLimit

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 `StmtCallLimitBehavior` property.

Data Source Methods

```
public Integer getStmtCallLimit()
```

```
public void setStmtCallLimit(Integer)
```

Default Value

`0`

Data Type

Integer

See also

StmtCallLimitBehavior on page 172

StmtCallLimitBehavior

Purpose

Specifies the behavior of the driver when the maximum web service call limit specified by the StmtCallLimit property is exceeded.

Valid Values

`ReturnResults | ErrorAlways`

Behavior

If set to `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 `ErrorAlways`, the driver generates an exception if the maximum web service call limit is exceeded.

Data Source Methods

```
public String getStmtCallLimitBehavior()
```

```
public void setStmtCallLimitBehavior(String)
```

Default Value

`ErrorAlways`

Data Type

String

See also

StmtCallLimit on page 171

Table

Purpose

Determines the name of the table your endpoint maps to when specifying an endpoint using the Sample property. 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.

Data Source Methods

```
public String getTable()
```

```
public void setTable(String)
```

Default Value

No default value

Data Type

String

See also

Sample on page 164

TokenURI

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 property. For example: `TokenURI=POST https://example.com/oauth2/authorize/`.
- See "OAuth 2.0 authentication" for more information.

Data Source Methods

```
public String getTokenUri()
```

```
public void setTokenUri(String)
```

Default Value

No default value

Data Type

String

See also

OAuth 2.0 authentication on page 84

TransactionMode

Purpose

Specifies how the driver handles manual transactions.

Valid Values

`NoTransactions` | `Ignore`

Behavior

If set to `NoTransactions`, the data source and the driver do not support transactions. Metadata indicates that the driver does not support transactions.

If set to `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.

Data Source Methods

```
public String getTransactionMode()
```

```
public void setTransactionMode(String)
```

Default Value

`NoTransactions`

Data Type

String

Truststore

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 property is ignored if ValidServerCertificate=`false`.

Data Source Methods

```
public String getTruststore()
```

```
public void setTruststore(String)
```

Default Value

No default value

Data Type

String

See also

Data Encryption on page 102

TrustStorePassword

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 property is ignored if ValidServerCertificate=`false`.

Data Source Methods

```
public String getTrustStorePassword()
```

```
public void setTruststorePassword(String)
```

Default Value

No default value

Data Type

String

See also

Data Encryption on page 102

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.

Data Source Methods

```
public String getUser()
```

```
public void setUser(String)
```

Default Value

No default value

Data Type

String

See also

Authentication on page 78

ValidateServerCertificate

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

`true` | `false`

If set to true, the driver validates the certificate that is sent by the REST service server. Any certificate from the server must be issued by a trusted CA in the truststore file. If the HostNameInCertificate property is specified, the driver also validates the certificate using a host name. The HostNameInCertificate property 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.

If set to false, the driver does not validate the certificate that is sent by the REST service server. The driver ignores any truststore information that is specified by the TrustStore and TrustStorePassword properties or Java system properties.

Notes

- Truststore information is specified using the TrustStore and TrustStorePassword properties or by using Java system properties.

Data Source Methods

```
public Boolean getValidateServerCert()
```

```
public void setValidateServerCert(Boolean)
```

Default Value

`true`

Data Type

Boolean

See also

Data Encryption on page 102

WSFetchSize

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.

Data Source Methods

```
public Integer getWSFetchSize()
```

```
public void setWSFetchSize(Integer)
```

Default Value

`2000`

Data Type

Integer

See also

Paging on page 197
Performance considerations on page 105

WSPoolSize

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 property. 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 WSPoolSize 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 property.

Data Source Methods

```
public Integer getWsPoolSize()
```

```
public void setWsPoolSize(Integer)
```

Default Value

`1`

Data Type

Integer

See also

Performance considerations on page 105

WSRetryCount

Purpose

Specifies the number of times the driver retries a timed-out Select request. The timeout period is specified by the WSTimeout connection property.

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.

Data Source Methods

```
public Integer getWSRetryCount()
```

```
public void setWSRetryCount(Integer)
```

Default Value

`5`

Data Type

Integer

WSTimeout

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 WSRetryCount is set to retry timed-out requests, the driver retries the request the specified number of times.

Data Source Methods

```
public Integer getWSTimeout()
```

```
public void setWSTimeout(Integer)
```

Default Value

`120`

Data Type

Integer

6

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 Autonomous REST Composer to generate a Model file, see Generating a Model file with the Autonomous REST Composer on page 59.

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 25: 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 183.
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 186.
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 188.
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 188.
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 191.
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 226.

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
206	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 26: 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 connection property. 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 property 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 property, 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 property. 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.

Table 27: 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 properties that specify confidential information, such as ClientID and ClientSecret, are not supported in the Model file. Values for these properties should be passed in a connection string or by the application.

```
"#authenticationMethod":"OAuth2"
"#authUri":"<auth_uri>"
"#logoffUri":"<log_off_uri>"
"#redirectUri":"<token_uri>"
"#scope":"<scope>"
"#tokenUri":"<token_uri>"
```

Table 28: 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
`#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 67 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. There is only one of these definitions per entry. 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 property 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 properties, you can create a variable by enclosing the property name in `{ }` brackets. For example, to reference the value of the password property in the connection string, specify `{password}`.

The exception to this behavior is the CustomAuthParams property. The value of the CustomAuthParams property is a semicolon-separated list of parameter values. To indicate the correct value in the list, in addition to the property 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 CustomAuthParams 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 property.
- 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 CustomAuthParams variables, if the specified ordinal position does not correlate to a value in the CustomAuthParams connection property, 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 properties 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 properties. 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 property value.

```
"#authentication": [
//Authorization header
    "HEADER Authorization=Basic (BASE64)({user}/{password})",
//Client ID value specified as the first argument of the customAuthParams connection
//property
    "HEADER X-Client-Id={customAuthParams[1]}",
//Client secret value specified as the second argument of the customAuthParams connection

//property
    "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 {/customAuthToken}"
]
```

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 for all the supported features and functionality.

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.

```
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     "<column4>[]":{"<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     "<column5>{<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     "<column6>":{"<nested_column1>":"<data_type>","<nested_column2>":"<data_type>"}
28     "<column7>":"<data_type>","<java_date_format>"
29     "<column8>":{"#type":"<data_type>","#extract:<reg_expression>"}
30     "<column9>":{"#type":"<data_type>","#header":true,"#eq":"<header_name>"}
31     "<column10>":{"#type":"<data_type>","#<operator>":"<uri_property>"}
32     }
33 }
```

Table 29: 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 194.
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 212.
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 195.

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 197.
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 205.
11	`#post`	Defines the sample values used when issuing a POST request. For details and syntax, see POST requests on page 206.
14	`#headers`	Specifies the HTTP headers to filter data returned by a GET request. For details and syntax, see Requests with custom HTTP headers on page 210.
15-30	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 216 and Data type mapping on page 217.
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 219.
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 219 and Nullable columns on page 220.
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 220.
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 221.
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 221.

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 222.
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 223.
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 223.
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 224.

See also

Query paths on page 212

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 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.

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 212

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` element. 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_x`

(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_x

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. Note that although it's possible to specify a single URL for the path, it is more typical that an array of URLs is necessary, as demonstrated here. 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 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}"]
          "#delete": "http://example.com/countries/{id}",
          "id":"VarChar(32)",
          "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)",
         "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 element (`#insert` | `#update`) after the `#key` element. In the following example, the primary key is passed in the body of insert and update operations because the `#insert` and `#update` elements 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 254
Update on page 265
Delete on page 253
Read-only columns on page 219
Query paths on page 212

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 30: 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.
`#postBodyPaging`	When retrieving results with a Select statement that uses a POST instead of a GET request, specifies whether paging parameters are sent as query parameters or sent in the body of the POST request. If set to `"false"` (the default), paging parameters are sent as query parameters. If set to `"true"`, paging parameters are sent in the body of the POST request. For example: `#postBodyPaging="true"`
`#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 31: 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 32: 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 204 for syntax examples.

Table 33: 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 34: 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 35: 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\": []}"
          }
     },
```

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 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.

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 188
Query paths on page 212

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"
      }
},
```

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 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.

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

Query paths on page 212
Requests with query parameters on page 215

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 ServerName connection property.

endpoint_path

is the path component of the URL endpoint. The value must be valid URL-encoded 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

URL-encoded values on page 246

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>",
```

where:

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, `countries`. The value must be valid URL-encoded 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

POST requests on page 206
URL-encoded values on page 246

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>,<java_date_format>}[/<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 ServerName property.

endpoint_path

is the path component of the URL endpoint. For example, `orders`. The value must be valid URL-encoded 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, `date`.

param_value

is the parameter value used for filtering the request during sampling. For example, `2020-07-01`.

java_date_format

(optional) is the format of your Date, Time, or Timestamp values using the Java SimpleDateFormat. For example, `yyyy-MM-dd`. For more information on the Java SimpleDateFormat, refer to https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html.

For example, the following demonstrates a GET request that will map to the `orders` table.

```
#path:"http://example.com/orders/get/{date:2020-07-01,yyyy-MM-dd}/all",
```

See also

POST requests on page 206
URL-encoded values on page 246

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 ServerName property.

endpoint_path

is the path component of the URL endpoint. For example, `times`. The value must be valid URL-encoded 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 246
POST requests on page 206

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 ServerName property.

endpoint_path

is the path component of the URL endpoint. For example, `times`. The value must be valid URL-encoded 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 246

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 36: 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[1]	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)[1]	Precision: 12 Minimum scale: 0 Maximum scale: 9
Timestamp(s)[1]	Precision: 23 Minimum scale: 0 Maximum scale: 9
TinyInt	Range: -999 to 999

[1] Formats for Data, Time, and Timestamp values are configurable. For more information, see Date, time, and timestamp formats on page 222.

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 39

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",
               "https://example.com/employees/{id}"]
           "#update":"http://example.com/countries/{id}",
           "employeeID":"VarChar(32),#key",
           "position_title":"VarChar(46),#readOnly",
           "start_year":"Integer"
           }
}
```

See also

Write operations on page 195

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)"}
```

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 37: 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
  ```

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 230 for more information and supported syntax.

`parameters`

Defines the parameter variables used in the function. See Parameters on page 229 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 232 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 231 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 234 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 230 for more information and supported syntax.

`parameters`

Defines the parameter variables used in the procedure. See Parameters on page 229 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 232 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 231 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 234 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 40

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 245 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 244 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 237 for more information.

`handler_declaration`

(optional) defines a handler used for exception handling. See Handler declarations on page 239 for more information.

`procedural_statement`

defines DQL, DML, DDL, and SQL procedural statements used to execute the routine. See Procedural SQL statements on page 235 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 234

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 245

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 242

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 241

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 237

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 230

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 230

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 38: 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"
        },
    }
```

7

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)

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 39: 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. Setting the Stmt_Call_Limit attribute has the same effect as setting the StmtCallLimit connection property. 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 260
Write operations on page 195

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 267
Select on page 255
Write operations on page 195

Refresh Map (EXT)

Purpose

The REFRESH MAP statement adds newly discovered objects to your relational view of native data. It also incorporates any configuration changes made to your relational view by reloading the schema definition and associated files.

Syntax

```
REFRESH MAP
```

Notes

- REFRESH MAP is an expensive query since it involves the discovery of native data.

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 0number`.

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 266

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 40: 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 274

Where clause

Purpose

Specifies the conditions that rows must meet to be retrieved.

Syntax

`WHERE` *`expr1rel_operatorexpr2`*

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 274
SQL expressions on page 266

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 274
SQL expressions on page 266

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 274
SQL expressions on page 266

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 266

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 274
Where clause on page 260
Write operations on page 195

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 41: 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'`
DATETIME	TIMESTAMP'*ts*'	`'2010-05-21 18:33:05.025'`

SQL Type	Literal Syntax	Example
DECIMAL	*n.f* where: *n* is the integral part *f* is the fractional part	0.25 3.1415 -7.48
DOUBLE	*n.f*E*x* 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 42: 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 43: 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 44: Comparison Operators

Operator	Purpose	Example
=	Equality test.	`SELECT * FROM emp WHERE sal = 1500`
!=<>	Inequality test.	`SELECT * FROM emp WHERE sal != 1500`

Operator	Purpose	Example
><	"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`
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 45: 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 46: Operator Precedence

Precedence	Operator
1	+ (Positive), - (Negative)
2	*(Multiply), / (Division)
3	+ (Add), - (Subtract)
4	|| (Concatenate)
5	=, >, <, >=, <=, <>, != (Comparison operators)
6	NOT, IN, LIKE

Precedence	Operator
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, including String, Numeric, Timedate, and System functions.

Refer to "Scalar functions" in the *Progress DataDirect for JDBC 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 47: 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]`

Condition	Description
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)
```

8

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 48: 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 49: 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 50: 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 51: 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 52: 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 53: 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 54: Response

Response	Data Type	Parameter Type	Description
status	INTEGER	OUT	The response code for the request.

getObjectSize

Purpose

Returns the size of an object.

Table 55: Required fields

Field Name	Data Type	Parameter Type	Description
bucket	VARCHAR(255)	IN	Specifies the name of the bucket containing the object for which the size is requested.
fileName	VARCHAR(255)	IN	Specifies the name of the object for which the size is requested.

Table 56: Response

Response	Data Type	Parameter Type	Description
size	BIGINT	OUT	The size of the specified object.

uploadFile

Purpose

Specifies the path of a local file to be uploaded.

Table 57: 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.
remoteFile	VARCHAR(255)	IN	Specifies the name of the file to create in the bucket.
localFile	VARCHAR(255)	IN	Specifies the path to a local file which contains the object.

downloadFile

Purpose

Creates a local file which contains the downloaded contents of an object.

Table 58: Required fields

Field Name	Data Type	Parameter Type	Description
`bucket`	VARCHAR(255)	IN	Specifies the name of the bucket containing the object to be downloaded.
`remoteFile`	VARCHAR(255)	IN	Specifies the name of the object to be downloaded.
`localFile`	VARCHAR(255)	IN	Specifies the path to a local file to be created to contain the retrieved data.

Box stored procedures

The following stored procedures are supported for Box cloud storage.

Table 59: 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 60: 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 61: 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 62: 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 63: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.

downloadObject

Purpose

Downloads the specified file.

Table 64: Required fields

Field Name	Data Type	Parameter Type	Description
`fileId`	VARCHAR(64)	IN	Specifies the ID of the file to be downloaded.

Table 65: 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 66: Required fields

Field Name	Data Type	Parameter Type	Description
`fileName`	VARCHAR(64)	IN	Specifies the name of the object to be uploaded.
`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 67: 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 68: 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 69: 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 70: 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 71: 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 72: 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 73: Required fields

Field Name	Data Type	Parameter Type	Description
`path`	VARCHAR(255)	IN	Specifies the destination name and path of the uploaded file.

Field Name	Data Type	Parameter Type	Description
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 74: 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 75: 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 76: 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 77: 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 78: 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 79: 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 80: 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.

Procedure Name	Description
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 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 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 82: 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 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 to which the file will be copied

Field Name	Data Type	Parameter Type	Description
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 84: 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 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
filePath	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.

createFolder

Purpose

Creates an empty folder in the specified location.

Table 87: Required fields

Field Name	Data Type	Parameter Type	Description
accountName	VARCHAR(64)	IN	Specifies the name of the storage account

Field Name	Data Type	Parameter Type	Description
`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 88: 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 89: 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 90: 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 91: Required fields

Field Name	Data Type	Parameter Type	Description
`contentLength`	INTEGER	IN	Specifies the length 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 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 92: 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 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
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 94: 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 95: 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 to which content is to be written. For example, `path/to/file`.

Table 96: Response

Response	Data Type	Parameter Type	Description
`status`	INTEGER	OUT	The response code for the request.