# Progress® DataDirect® Autonomous REST Connector *for* JDBC™

## User's Guide for Partners

***Release 6.0.0***

# Copyright

**Updated: 2020/06/26**

# 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-only access to JSON-based REST API data sources. To support SQL access to REST services, the driver creates a relational map of the returned JSON 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 that allows you to modify and persist a map. In addition, the driver employs a SQL engine component that provides support for 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. See "Setting up the driver" for an overview of the recommended setup of the driver.

**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?

### Changes Since 6.0.0 Release

- **Driver Enhancements**
  - 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 input REST file to define a custom authentication flow. See Custom authentication on page 49 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 input REST 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 119 for details.
  - The driver has been enhanced to support OAuth 2.0 authentication. See OAuth 2.0 authentication on page 44 for details.
  - The driver has been enhanced to support requests for endpoints that use custom HTTP-headers. See Creating an input REST file on page 20 for details.

### 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 149 for details.
- The driver supports standard JSON data types and additional data types through data type inference.
- The driver supports using internal memory or a configurable REST file to define REST responses and relational mapping. See Configuring the relational map on page 18 for details.
- The driver heuristically maps data types, eliminating the need to define native data types in most scenarios.
- The driver supports basic, HTTP-header based, URL-Parameter based and no authentication. See Authentication on page 42 for details.
- The driver supports the handling of large result sets with configurable paging and the FetchSize on page 79 and WSFetchSize on page 112 connection properties. See Configuring the relational map on page 18 for more information on configuring paging.

## Setting up the driver

After installation, you will need to complete several tasks before you can begin accessing data with the Autonomous REST Connector. This section provides you with an overview of those tasks and the recommended set-up of the driver. Refer to the references within the steps for detailed information related to each task.

**To begin accessing data with the driver:**

1. Determine the method the driver will use to sample endpoints for relational mapping:
   - Using the Sample property: You can specify a single endpoint to sample using the Sample property. At connection, the driver samples the specified endpoint and infers a schema based on the results. This method allows you to begin accessing data with a minimal amount of configuration, but it lacks some of the functionality supported by the input REST file method.

See Using the Sample property method on page 19 for more information. Skip to Step 6 on page 11.

- Using the input REST file: You can specify multiple endpoints to sample using the input REST file. This method also allows you to leverage the file's supported syntax to define POST requests, and configure paging and other customizations supported by the input REST file.

  See Using the input REST file method on page 20 for more information. Proceed to the next step.

You must use an input REST file if your session does any of the following; otherwise, we recommend using the Sample property:

- Accesses multiple endpoints
- Issues POST requests
- Accesses endpoints that require paging
- Uses a custom authentication method
- Uses other customizations supported by the input REST file

2. Using a text editor, create an input REST file. The input REST file is a simple text file that uses the `file_name.rest` naming convention.
3. In the REST file, type a comma-separated list of the GET endpoints to be used by your session. For example:

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

   See Input REST file syntax on page 117 for a detailed description of the syntax supported by the file.

4. If your session uses POST requests, type your POST endpoints in the comma-separated list of endpoints in your input REST file. See POST requests on page 131 for details.
5. If any of your endpoints require paging, configure either offset or page numbering paging for affected endpoints in your input REST file. See Paging on page 129 for details.
6. Configure the driver to connect using connection URLs or data sources as described in Connecting from an application on page 23. The following groups of properties should be addressed:

   a) **Required properties:** Configure the required properties based on whether you are using the sample property or input REST file:

   - **Sample property:** Set the Sample property to specify the endpoint to which you want to connect and sample. For example, `https://example.com/countries/`.
   - **Input REST file:** Set the Config property to specify the name and location of the input REST file. For example, `C:\path\to\myrest.rest` (Windows) or *`home_dir`*`/path/to/myrest.rest` (UNIX/Linux).

   b) **Authentication method properties:** Configure the driver according to the authentication method used by your REST service. The driver supports the following methods:

   - **No Authentication**: The driver does not attempt to authenticate.
   - **Basic Authentication**: The driver authenticates using the specified user IDs, passwords, and HTTP headers.
   - **HTTP Header Authentication**: The driver passes security tokens via the HTTP headers to authenticate. In some scenarios, the REST services may also authenticate the user ID.

- **URL Parameter Authentication**: The driver authenticates by passing security tokens using URLs. In some scenarios, the REST services may also authenticate the user ID.
- **OAuth 2.0 Authentication**: The driver authenticates using OAuth 2.0 authentication flows.
- **Custom authentication requests**: The driver uses a custom token-based authentication flow that is defined in the input REST file.

See Authentication on page 42 for configuration details.

c) **Optional properties:** Set the values for any optional properties that you want to configure. See Using connection properties on page 24 for a complete list of supported properties by functionality.

7. Connect to your REST service.

This completes the deployment of the driver.

## Connection URL

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 REST file.

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

```
jdbc:subprotocol:autorest://servername;Config=rest_file_path;[property=value[;...]];
```

For sessions using the Sample property:

```
jdbc:subprotocol:autorest:Sample=sample_path;[property=value[;...]];
```

where:

*subprotocol*

is the portion of the connection URL subprotocol that is determined by the publisher of your application. For this value, refer to your application's documentation.

*servername*

optionally, specifies the host name portion of the URL 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.

*rest_file_path*

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

*property=value*

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

*sample_path*

specifies the endpoint to sample when not using an input REST file.

The following examples demonstrate URLs for sessions using no authentication:

For sessions using an input REST file:

```
jdbc:*subprotocol*:autorest:https://example.com/;Config=C:/path/to/myrest.rest;
```

For sessions using the Sample property:

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

**Note:** Connection URLs containing special characters should be enclosed in quotes; otherwise, the connection may fail.

### See also

Connection property descriptions on page 63

## 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 REST file is loaded and/or any of the initial sampling has been completed. 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`*.

---

The primary key for parent tables are determined heuristically from the top-level fields in the document (see "Determining the primary key"). For example, `resident_id`. If necessary, you can designate a new primary key in a parent table by editing the resolved REST file. For details, see "Designating a primary key."

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 16
Designating the primary key on page 21
Table on page 107

## 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"},
 "1979-11-06":{"attendance":"34000","opponent":"Mustangs","result":"loss"},
 "1979-11-06":{"attendance":"8500","opponent":"Jets","result":"loss"},
}
```

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

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. 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 |
| 1979-11-06 | 34000 | Mustangs | loss |
| 1979-11-06 | 8500 | Jets | loss |

## 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 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 no top-level column is available, nested columns inside objects may be considered
- If the search runs out of candidates, a best-case candidate will be selected

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.

2

# Using the driver

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

For details, see the following topics:

## Configuring the relational map

The Autonomous REST Connector maps JSON responses to a relational model, exposing your REST data to relational, SQL-based applications. The driver supports two methods for mapping data: using the Sample property to map a single endpoint, and using an input REST file to define one or more endpoints. Which method you use depends on the characteristics of your session. If any of the following apply to your session, you will need to create an input REST file:

- 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

If none of these characteristics apply, you can use either method.

**Sample property method**

When using the Sample property, the driver issues a query to the endpoint specified with the Sample property at connection. The results of the query provide a sample of the data, which are then inspected by the driver and used to infer a schema. Using the Sample property method requires minimal configuration, but offers limited functionality compared to the input REST file method. For more information on configuring this method, see "Using the Sample property method."

**Input REST file method**

In addition to being able to specify multiple endpoints for your session, the input REST file method also allows you to define POST requests, configure paging, and define additional customizations described in "Creating an input REST file." To use this method, you will need to create an input REST file, which is a simple text file that contains a comma separated list of endpoints. After you create the file, you will need to specify its location using the Config property.

See "Creating an input REST file" for a full description of the syntax used in the REST file. For information on configuring the driver for the input REST file method, see "Using the input REST file method".

**Schema map generation**

Upon initial connection, the driver generates a schema map that is stored in either internal memory or a resolved REST file--depending on the setting of the CreateMap property. When CreateMap is set to `Session` (the default), the driver stores the schema map using internal memory, which persists for only the life of the session before being discarded. When CreateMap is set to `forceNew` or `notExist`, the driver generates the relational map in a resolved REST file and set of internal driver files. The REST file persists indefinitely when the property is set to `notExist`. Conversely, if the property is set to `forceNew`, the driver deletes and regenerates the REST file at the beginning of every connection. See "CreateMap" for details.

**Resolved REST file**

The resolved REST file is a driver generated file that contains the fully defined map of REST responses. The driver uses this map to execute SQL queries. The resolved REST file is distinct from the input REST file, which is user created and supplies a list of end point for the driver to sample. In other words, the input REST file specifies which endpoints to sample, while the resolved REST file stores the response definitions discovered by sampling. Note that the resolved REST file is supported using both the Sample property and input REST file methods.

The driver generates the resolved REST file in the location specified by the SchemaMap property. In most scenarios, the resolved REST file works transparently from this location, with no additional driver configuration. However, in some instances, you may want to change the relational view generated by the driver. To modify the relational view, you will need to edit the resolved REST file. See "Modifying the relational view" for details.

### See also

Using the Sample property method on page 19
Using the input REST file method on page 20
CreateMap on page 74
Modifying the relational view on page 21
Mapping objects to tables on page 13

## Using the Sample property method

The sample property method allows you to access data for a single endpoint with minimal configuration.

To configure the driver to using the Sample property method:

- Set the Sample property to specify the endpoint to which you want to connect and sample. For example, `https://example.com/countries/events/1`.
- Optionally, set the Table property to specify the name of the table your endpoint maps to in the relational view of the data. If you do not specify a value, the driver will automatically generate a name based on the composition of the endpoint. For example, the following endpoint:

  ```
  https://example.com/countries/events/1/
  ```

  maps to the following table name:

  ```
  COUNTRIES_EVENTS_1
  ```

- Optionally, if you want to store the relational map in a resolved REST file, set the CreateMap property to either of the following values:
  - `forceNew`: The driver deletes the current REST file, internal configuration files, and relational map in the location specified by the SchemaMap property and creates a new set at the same location.
  - `notExist`: The driver uses the current REST file, internal files, and relational map in the location specified by the SchemaMap property. If the files do not exist, the driver creates them.

  Note that if no value is specified for CreateMap, the driver defaults to using internal memory to store the map (`CreateMap=session`).

- Optionally, if you are storing the relational map to a REST file (CreateMap=`forceNew` | `notExist`) , you can set the SchemaMap property to specify the location to generation the REST and internal driver files.
- Optionally, set the values for any required authentication properties. See "Authentication" for details.

For example, the following connection string demonstrates the minimum connection properties to configure the Sample property method:

```
jdbc:subprotocol:autorest:Sample=https://example.com/countries/events/1;
```

**Note:** Connection URLs containing special characters should be enclosed in quotes; otherwise, the connection may fail.

## Using the input REST file method

The input REST file method allows you to access data from one or more endpoints, define POST requests, and configure additional customizations supported by the input REST file.

To configure the driver to using the input REST file property method:

- Set the Config property to specify the name and location of the input REST file used to define your endpoints for sampling. For example, `C:\path\to\myrest.rest` (Windows) or `*home_dir*/path/to/myrest.rest` (UNIX/Linux).
- Optionally, if you want to store the relational map in a resolved REST file, set the CreateMap property to either of the following values:
  - `forceNew`: The driver deletes the current REST file, internal configuration files, and relational map in the location specified by the SchemaMap property and creates a new set at the same location.
  - `notExist`: The driver uses the current REST file, internal files, and relational map in the location specified by the SchemaMap property. If the files do not exist, the driver creates them.

  Note that if no value is specified for CreateMap, the driver defaults to using internal memory to store the map (`CreateMap=session`).
- Optionally, if you are storing the relational map to a REST file (CreateMap=`forceNew` | `notExist`) , you can set the SchemaMap property to specify the location to generation the REST and internal driver files.
- Optionally, set the values for any required authentication properties. See "Authentication" for details.

For example, the following connection string demonstrates the minimum connection properties to configure

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

**Note:** Connection URLs containing special characters should be enclosed in quotes; otherwise, the connection may fail.

### See also

Authentication on page 42

## Creating an input REST file

The input REST file is a simple text file that uses the *file_name*`.rest` naming convention. To configure the file, you will need to populate the contents with a list of comma-separated endpoints and requests using the formats described in "Input REST file syntax". For additional examples, see "Example input REST file."

In addition to mapping endpoints to tables, you can customize your mapping and driver behavior using the REST file. See "Input REST file syntax" for details.

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

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

### See also

Input REST file syntax on page 117

Example input REST file  on page 147

## Modifying the relational view

**Note:** You can modify the relational view only if your session uses REST files. For sessions that use internal memory to map the relational view (`CreateMap=Session`), the following functionality is not supported.

You can manually edit the resolved REST file to modify the relational view of your data, including adding/removing endpoints or updating the primary key. When you first issue a SQL command for a session, the driver overwrites the resolved REST file to the location specified by the SchemaMap. To persist your changes, you will need to move the edited REST file to the location specified by the Config property. This procedure will guide you through the process of modifying the REST file.

**To modify your REST file:**

1. Copy the REST file in the location specified by the SchemaMap property to a new location or the current location specified by the Config property. Note, if moving the file to a different location, the driver must have read access to the new directory.
2. Edit the contents of the copied REST file to do either of the following:
   - Designate the primary key for a table. See Designating the primary key on page 21 for details.
   - Add/remove endpoints from the map. See Adding/removing objects on page 22 for details.
3. Set the Config property to specify the name and location of the edited REST file.
4. Set the CreateMap property to `ForceNew`.

   **Note:** After completing this procedure, set the CreateMap property to your preferred setting.

5. Use the driver to execute any SQL command to regenerate the resolved REST file.

The resolved REST file, and therefore your relational view, now includes your modifications. If you need to make additional modifications, repeat the steps discussed in this procedure.

### Designating the primary key

You can designate the primary key for a table by modifying the REST file. In the column object, add the `#key` after the data type element, separated by a comma. 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",
           }
}
```

Your changes will be updated during your next connection. See "Modifying the relational view" for details on regenerating the REST file to include your changes.

### See also

Modifying the relational view on page 21

## Adding/removing objects

After you generate your initial relational map, you can add endpoints or remove objects at any time by editing the REST file. The following sections provide you with detailed instructions.

### Adding endpoints

To add endpoints to your map, add a comma-separated entry for each endpoint using the following format. This process is similar to the one used to create the input REST file.

```
"<table_name1>":"<endpoint1>",
```

For example, the following demonstrates adding an endpoint for `new_table` table to a REST file containing a resolved entry `my_table` table.

```
{
"new_table":"http://example.com/countries/",
"my_table":{
          "#path":[
              "https://example.com/employees"
          ],
          "employeeID":"VarChar(32),#key",
          "position_title":"VarChar(46)",
          "start_year":"Integer",
           }
}
```

The added table or tables will be resolved during the next connection. See "Modifying the REST file" for details on regenerating the REST file to include your changes.

### Removing objects from the relational view

To remove an object, delete the resolved table entry or object from the REST file. See "Modifying the REST file" for details on regenerating the REST file to include your changes.

### See also

Modifying the relational view on page 21

## Mapping new native objects to a table

You can map new JSON objects to an existing relational table by configuring the driver to resample the table. To resample a table, set the Table property to specify the name of the table you want to update. The next time the driver connects to the data source, the driver will resample the table and map any new JSON objects. Note that any objects removed from the native view will persist after resampling. To remove these objects, see "Adding/removing objects".

See “Table” for more information on the Table property.

### See also

Adding/removing objects on page 22

Table on page 107

# Connecting from an application

After the driver has been installed and defined on your class path, you can connect from your application to your data using a connection URL.

## Passing the connection URL

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 REST file.

### Connection URL Syntax

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

```
jdbc:subprotocol:autorest://servername;Config=rest_file_path;[property=value[;...]];")
```

For sessions using the Sample property:

```
jdbc:subprotocol:autorest:Sample=sample_path;[property=value[;...]];
```

where:

*servername*

optionally, specifies the host name portion of the URL 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.

*rest_file_path*

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

### Connection URL Example

For sessions using a REST file:

```
jdbc:subprotocol:autorest:https://example.com/;Config=C:\path\to\myrest.rest;
```

For sessions using the Sample property:

```
jdbc:subprotocol:autorest:Sample='https://example.com//countries/get/all';
```

### See also

Config on page 73
Sample on page 99
Connection property descriptions on page 63

## Using connection properties

You can use connection properties to customize the driver for your environment. This section organizes connection properties according to functionality. Connection properties are specified in your connection URL as a key value pair and takes the form `property=value`.

See "Connection property descriptions" for an alphabetical list of connection properties and their descriptions.

### See also

Passing the connection URL on page 23
Connection property descriptions on page 63

## Required properties

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

**Table 5: Required properties**

| Property | Characteristic |
|---|---|
| Config on page 73 | Specifies the name and location of the input REST 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 an input REST file if your session:<br>• Accesses multiple endpoints<br>• Issues POST requests<br>• Accesses endpoints that require paging<br>• Accesses endpoints that use custom HTTP headers<br>• Uses custom HTTP response code processing<br>• Requires a custom authentication flow |
| Sample on page 99 | 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 an input REST file.<br>**Note:** This property is required when not using an input REST file. |

## Mapping properties

The following table summarizes connection properties involved in mapping the REST API data model to a local schema map used to support SQL queries.

**Table 6: Mapping properties**

| Property | Characteristic |
|---|---|
| CreateMap on page 74 | Determines whether the driver creates the internal files required for a relational map of the native data when establishing a connection.<br>If set `session`, the driver uses memory to store the internal configuration information and relational map of native data. A REST file is not created when this value is specified. After the session, the view is discarded.<br>If set to `forceNew`, the driver deletes the current REST file, internal configuration files, and relational map in the location specified by the SchemaMap property and creates a new set at the same location.<br>If set to `notExist`, the driver uses the current REST file, internal files, and relational map in the location specified by the SchemaMap property. If the files do not exist, the driver creates them.<br>The default is `session`. |

| Property | Characteristic |
|---|---|
| RefreshSchema on page 96 | Specifies whether the driver automatically refreshes the map of the data model when a user connects to a REST service.<br>If set to `true`, the driver automatically refreshes the map of the data model when a user connects to a REST service. Changes to objects since the last time the map was generated will be shown in the metadata.<br>If set to `false`, the driver does not refresh the map of the data model when a user connects to a REST service.<br>The default is `true`. |
| SchemaMap on page 100 | Specifies the directory where the internal configuration files, REST file, and the relational map of the REST data model are written. The driver looks for these files when connecting to a REST service. If the file does not exist, the driver creates one.<br>The default varies based on your environment:<br>For Windows:<br>*`application_data_folder`*`\Local\Progress\DataDirect\AutoREST_Schema\`<br>For UNIX/Linux:<br>`~/progress/datadirect/AutoREST_schema/` |
| ServerName on page 103 | Specifies the host name portion of the URL endpoint to which you send requests. This property allows you to define endpoints without storing the host name component in the REST file property. |
| Table on page 107 | 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 an input REST file, the driver will resample the endpoint associated with this table and add any newly discovered columns to the relational view. |

### See also

Mapping objects to tables on page 13

## Basic authentication properties

The following table summarizes connection properties which are required for basic authentication.

**Table 7: Authentication properties**

| Property | Characteristic |
|---|---|
| AuthenticationMethod on page 69 | Determines which authentication method the driver uses during the course of a session.<br><br>If set to `None`, the driver does not attempt to authenticate.<br><br>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).<br><br>If set to `Custom`, the driver uses the custom token-based authentication flow that is defined in the input REST file.<br><br>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.<br><br>If set to `OAuth2`, the driver uses OAuth 2.0 to authenticate to REST endpoints.<br><br>If set to `UrlParameter`, the driver passes security tokens via the URL for authentication. You must also configure the AuthParam and SecurityToken properties.<br><br>The default is `None`. |
| AuthHeader on page 70 | Specifies the name of the HTTP header used for authentication. This property is used when Basic (`AuthenticationMethod=Basic`) or Header-based token authentication (`AuthenticationMethod=HttpHeader`) is enabled; otherwise, this property is ignored.<br><br>The default is `Authorization`. |
| Password on page 90 | Specifies the password to use to connect to your REST service. This property is ignored when `AuthenticationMethod=None`. |
| User on page 110 | Specifies the user name that is used to connect to the REST service. A user name is required if user is enabled by your REST service. This property is ignored when `AuthenticationMethod=None`. |

## HTTP header authentication properties

The following table summarizes connection properties which are required for HTTP header authentication.

**Table 8: Authentication properties**

| Property | Characteristic |
|---|---|
| AuthenticationMethod on page 69 | Determines which authentication method the driver uses during the course of a session.<br><br>If set to `None`, the driver does not attempt to authenticate.<br><br>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).<br><br>If set to `Custom`, the driver uses the custom token-based authentication flow that is defined in the input REST file.<br><br>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.<br><br>If set to `OAuth2`, the driver uses OAuth 2.0 to authenticate to REST endpoints.<br><br>If set to `UrlParameter`, the driver passes security tokens via the URL for authentication. You must also configure the AuthParam and SecurityToken properties.<br><br>The default is `None`. |
| AuthHeader on page 70 | Specifies the name of the HTTP header used for authentication. This property is used when Basic (`AuthenticationMethod=Basic`) or Header-based token authentication (`AuthenticationMethod=HttpHeader`) is enabled; otherwise, this property is ignored.<br><br>The default is `Authorization`. |
| SecurityToken on page 102 | Specifies the security token required to make a connection to your REST API endpoint. This property is required when token based authentication is enabled (`AuthenticationMethod=HttpHeader` \| `UrlParameter`); otherwise, this property is ignored. 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. |

## OAuth 2.0 properties

The following table summarizes connection properties that are used for OAuth 2.0 authentication. See "OAuth 2.0 authentication" for the properties used for each authentication flow.

**Table 9: Authentication Properties**

| Property | Characteristic |
|---|---|
| AccessToken on page 68 | Specifies the access token required to authenticate to a REST service when OAuth 2.0 is enabled (`AuthenticationMethod=OAuth2`). Typically, this property is configured by the application |

| Property | Characteristic |
|---|---|
| AuthenticationMethod on page 69 | Determines which authentication method the driver uses during the course of a session.<br>If set to `None`, the driver does not attempt to authenticate.<br>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).<br>If set to `Custom`, the driver uses the custom token-based authentication flow that is defined in the input REST file.<br>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.<br>If set to `OAuth2`, the driver uses OAuth 2.0 to authenticate to REST endpoints.<br>If set to `UrlParameter`, the driver passes security tokens via the URL for authentication. You must also configure the AuthParam and SecurityToken properties.<br>The default is `None`. |
| ClientID on page 71 | Specifies the client ID key for your application. The driver uses this value for certain flows when authenticating to a REST service using OAuth 2.0 (`AuthenticationMethod=OAuth2`). |
| ClientSecret on page 72 | Specifies the client secret for your application when authenticating to a REST service with OAuth 2.0 enabled (`AuthenticationMethod=OAuth2`).<br>**Important:** The client secret is a confidential value used to authenticate the application to the server. The value must be securely maintained to prevent unauthorized access. |
| LogoffURI on page 87 | 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. |
| OAuthCode on page 89 | Specifies the temporary authorization code that is exchanged for access tokens when OAuth 2.0 authentication is enabled (`AuthenticationMethod=OAuth2`). Authorization codes are used to authenticate against the endpoint specified by the TokenURI property. If authentication is successful, an access token is generated and fetched from the specified location. |
| Password on page 90 | Specifies the password to use to connect to your REST service. |
| RedirectURI on page 95 | Specifies the endpoint to which the client is returned after authenticating with a third-party service when OAuth 2.0 authentication is enabled (`AuthenticationMethod=OAuth2`). |

| Property | Characteristic |
|---|---|
| Scope on page 101 | Specifies a space-separated list of OAuth scopes that limit the permissions granted by an access token. The driver uses this value when authenticating to a REST service using OAuth 2.0 (`AuthenticationMethod=OAuth2`). |
| TokenURI on page 108 | Specifies the endpoint used to exchange authentication credentials for access tokens when OAuth 2.0 authentication is enabled (`AuthenticationMethod=OAuth2`). |
| User on page 110 | Specifies the user name that is used to connect to the REST service. A user name is required if user is enabled by your REST service. |

### See also

OAuth 2.0 authentication on page 44

## URL parameter authentication properties

The following table summarizes connection properties which are required for URL parameter authentication.

**Table 10: Authentication Properties**

| Property | Characteristic |
|---|---|
| AuthenticationMethod on page 69 | Determines which authentication method the driver uses during the course of a session.<br><br>If set to `None`, the driver does not attempt to authenticate.<br><br>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).<br><br>If set to `Custom`, the driver uses the custom token-based authentication flow that is defined in the input REST file.<br><br>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.<br><br>If set to `OAuth2`, the driver uses OAuth 2.0 to authenticate to REST endpoints.<br><br>If set to `UrlParameter`, the driver passes security tokens via the URL for authentication. You must also configure the AuthParam and SecurityToken properties.<br><br>The default is `None`. |
| AuthParam on page 70 | 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. |

| Property | Characteristic |
|---|---|
| SecurityToken on page 102 | Specifies the security token required to make a connection to your REST API endpoint. This property is required when token based authentication is enabled (`AuthenticationMethod=HttpHeader` \| `UrlParameter`); otherwise, this property is ignored. 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. |
| User on page 110 | Specifies the user name that is used to connect to the REST service. A user name is required if user is enabled by your REST service. This property is ignored when `AuthenticationMethod=None`. |

## Custom authentication request properties

The following table summarizes connection properties which are used for custom authentication requests that defined in the input REST file.

**Table 11: Authentication Properties**

| Property | Characteristic |
|---|---|
| AuthenticationMethod on page 69 | Determines which authentication method the driver uses during the course of a session.<br>If set to `None`, the driver does not attempt to authenticate.<br>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).<br>If set to `Custom`, the driver uses the custom token-based authentication flow that is defined in the input REST file. See "Custom authentication requests" for more information on input REST file.<br>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.<br>If set to `OAuth2`, the driver uses OAuth 2.0 to authenticate to REST endpoints.<br>If set to `UrlParameter`, the driver passes security tokens via the URL for authentication. You must also configure the AuthParam and SecurityToken properties.<br>The default is `None`. |
| CustomAuthParams on page 76 | Specifies a list of parameter values used by custom authentication requests that are defined in the input REST file. This property allows you to configure parameter values used in custom authentication requests on a per connection basis, without editing the REST file, and securely pass them in a connection string. |

| Property | Characteristic |
|---|---|
| User on page 110 | Specifies the user name that is used to connect to the REST service. A user name is required if user is enabled by your REST service. This property is ignored when `AuthenticationMethod=None`. |
| Password on page 90 | Specifies the password to use to connect to your REST service. |

### See also

Custom authentication requests on page 124
Authentication on page 42

## Data encryption properties

The following table summarizes connection properties which can be used to enable SSL.

**Table 12: Data encryption properties**

| Property | Characteristic |
|---|---|
| CryptoProtocolVersion on page 75 | Specifies a cryptographic protocol or comma-separated list of cryptographic protocols that can be used when SSL is enabled (`EncryptionMethod=SSL`). |
| EncryptionMethod on page 78 | Determines whether data is encrypted and decrypted when transmitted over the network between the driver and REST service.<br>If set to `noEncryption`, data is not encrypted or decrypted.<br>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.<br>The default is `noEncryption`.<br>**Note:** SSL encryption is enabled when the URL specified in the Sample property or REST file uses HTTPS, regardless of the setting of EncryptionMethod. |
| HostNameInCertificate on page 80 | Specifies a host name for certificate validation when SSL encryption is enabled (`EncryptionMethod=SSL`) 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. |
| KeyPassword on page 84 | Specifies the password that is used to access the individual keys in the keystore file when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the REST server. This property is useful when individual keys in the keystore file have a different password than the keystore file. |

| Property | Characteristic |
|---|---|
| KeyStore on page 84 | Specifies the directory of the keystore file to be used when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the REST server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request. |
| KeyStorePassword on page 85 | Specifies the password that is used to access the keystore file when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the REST server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request. |
| TrustStore on page 109 | Specifies the directory of the truststore file to be used when SSL is enabled (`EncryptionMethod=SSL`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts. |
| TrustStorePassword on page 110 | Specifies the password that is used to access the truststore file when SSL is enabled (`EncryptionMethod=SSL`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts. |
| ValidateServerCertificate on page 111 | Determines whether the driver validates the certificate that is sent by the database server when SSL encryption is enabled (`EncryptionMethod=SSL`). When using SSL server authentication, any certificate that is sent by the server must be issued by a trusted Certificate Authority (CA).<br>The default is `true`. |

### See also

Data encryption on page 50

## Proxy server properties

The following table summarizes proxy server connection properties.

**Table 13: Proxy Server Properties**

| Property | Characteristic |
|---|---|
| ProxyHost on page 91 | Identifies a proxy server to use for the first connection. This can be a named server or an IP address. |
| ProxyPassword on page 92 | Specifies the password needed to connect to a proxy server for the first connection. |
| ProxyPort on page 93 | Specifies the port number where the proxy server is listening for HTTP or HTTPS requests for the first connection.<br>The default is `0`. |
| ProxyUser on page 93 | Specifies the user name needed to connect to a proxy server for the first connection. |

### See also

- Connecting through a proxy server on page 40

## Web service properties

The following table summarizes Web service connection properties, including those related to timeouts.

**Table 14: Web Service Properties**

| Property | Characteristic |
|---|---|
| LoginTimeout on page 87 | The amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request.<br>If set to `0`, the driver does not time out a connection request.<br>If set to *x*, the driver waits for the specified number of seconds before returning control to the application and throwing a timeout exception.<br>The default is `0` (no timeout). |
| StmtCallLimit on page 106 | Specifies the maximum number of Web service calls the driver can make when executing any single SQL statement or metadata query.<br>If set to `0`, there is no limit.<br>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.<br>The default is `0` (no limit). |
| StmtCallLimitBehavior on page 107 | Specifies the behavior of the driver when the maximum Web service call limit specified by the StmtCallLimit property is exceeded.<br>If set to `errorAlways`, the driver generates an exception if the maximum Web service call limit is exceed.<br>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.<br>The default is `errorAlways`. |

| Property | Characteristic |
|---|---|
| WSFetchSize on page 112 | Specifies the number of rows of data the driver attempts to fetch for each JDBC call when paging is enabled for an endpoint.<br>If set to `0`, the driver attempts to fetch up to the maximum number of row specified by the maximumPageSize property. This value typically provides the maximum throughput.<br>If set to *x*, the driver attempts to fetch up to the 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 WSFetch size for interactive applications only.<br>`0` (up to the maximum number of rows specified by the maximumPageSize property) |
| WSPoolSize on page 113 | 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.<br>The default is `1`. |
| WSRetryCount on page 114 | The number of times the driver retries a timed-out Select request.<br>If set to `0`, the driver does not retry timed-out requests after the initial unsuccessful attempt.<br>If set to `x`, the driver retries the timed-out request the specified number of times.<br>The default is `5`. |
| WSTimeout on page 115 | Specifies the time, in seconds, that the driver waits for a response to a Web service request.<br>If set to `0`, the driver waits indefinitely for a response; there is no timeout.<br>If set to *x*, the driver uses the value as the default timeout for any statement created by the connection.<br>The default is `120`. |

## Data type handling properties

The following table summarizes connection properties which can be used to handle data types.

**Table 15: Data type handling properties**

| Property | Characteristic |
|---|---|
| ConvertNull on page 74 | Controls how data conversions are handled for null values.<br>If set to `0`, 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.<br>If set to `1`, 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.<br>The default is `1`. |
| JDBCBehavior on page 83 | Determines how the driver describes native data types that map to the following JDBC 4.0 data types: NCHAR, NVARCHAR, NLONGVARCHAR, NCLOB, and SQLXML.<br>If set to `0`, the driver describes the data types as JDBC 4.0 data types when using Java SE 8 or higher.<br>If set to `1`, the driver describes the data types using JDBC 3.0-equivalent data types, regardless of JVM. This allows your application to continue using JDBC 3.0 types in a Java SE 8 or higher environment.<br>The defaut is `1`. |

## Timeout properties

The following table summarizes timeout connection properties.

**Table 16: Timeout Properties**

| Property | Characteristic |
|---|---|
| LoginTimeout on page 87 | The amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request.<br>If set to `0`, the driver does not time out a connection request.<br>If set to *x*, the driver waits for the specified number of seconds before returning control to the application and throwing a timeout exception.<br>The default is `0` (no timeout). |

| Property | Characteristic |
|---|---|
| WSRetryCount on page 114 | The number of times the driver retries a timed-out Select request.<br>If set to `0`, the driver does not retry timed-out requests after the initial unsuccessful attempt.<br>If set to `x`, the driver retries the timed-out request the specified number of times.<br>The default is `0`. |
| WSTimeout on page 115 | Specifies the time, in seconds, that the driver waits for a response to a Web service request.<br>If set to `0`, the driver waits indefinitely for a response; there is no timeout.<br>If set to *x*, the driver uses the value as the default timeout for any statement created by the connection.<br>The default is `120`. |

## Statement pooling properties

The following table summarizes statement pooling connection properties.

**Table 17: Statement Pooling Properties**

| Property | Characteristic |
|---|---|
| ImportStatementPool on page 81 | 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. |

| Property | Characteristic |
|---|---|
| MaxPooledStatements on page 88 | 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.<br>If set to `0`, the driver's internal prepared statement pooling is not enabled.<br>If set to *x*, the driver enables the DataDirect Statement Pool and uses the specified value to cache a certain number of 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.<br>The default is `0`. |
| RegisterStatementPoolMonitorMBean on page 98 | 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.<br>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.<br>If set to `false`, the driver does not register an MBean for the Statement Pool Monitor for any statement pool.<br>The default is `false`. |

Refer to "Statement Pool Monitor" in the *Progress DataDirect for JDBC Drivers Reference* for an overview of statement pooling.

## Additional properties

The following table summarizes additional connection properties.

**Table 18: Additional Properties**

| Property | Characteristic |
|---|---|
| DebugRecord on page 77 | Specifies the directory where the driver generates debug record files. When a value is specified, the driver records server requests and responses to 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.<br><br>**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. |
| FetchSize on page 79 | 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.<br><br>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.<br><br>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.<br><br>The default is `100` (rows). |
| InsensitiveResultSetBufferSize on page 82 | Determines the amount of memory used by the driver to cache insensitive result set data.<br><br>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.<br><br>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.<br><br>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.<br><br>The default is `2048`. |

| Property | Characteristic |
|---|---|
| LogConfigFile on page 86 | Specifies the filename of the configuration file used to initialize driver logging.<br>The default is `ddlogging.properties`. |
| ReadAhead on page 94 | Specifies the maximum number of fetch requests the driver issues in parallel. By default, the driver queues the next page while processing the current page. This property allows you to fetch multiple requests simultaneously, thereby improving throughput and performance.<br>**Caution:** Due to potential impacts to other users, 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. |
| RefreshDirtyCache on page 96 | 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.<br>If set to `1`, 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.<br>If set to `0`, a dirty cache is not refreshed when the cache is referenced in a fetch operation.<br>The default is `1`. |
| SpyAttributes on page 103 | Enables DataDirect Spy to log detailed information about calls issued by the driver on behalf of the application. DataDirect Spy is not enabled by default. |

## 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 63

## 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.

## 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.
- *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 input REST file.

By default, the driver is configured to use no authentication (`AuthenticationMethod=None`).

### See also

AuthenticationMethod on page 69

## Basic authentication

To configure the driver to use basic authentication:

- Configure the minimum properties required for a connection:
  - If you are using a REST file, set the Config property to provide the name and location of the input REST 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 your logon ID.
- Set the Password property to specify your password.
- Optionally, specify values for any additional properties you want to configure.

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

For a connection URL:

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

### See also

Basic authentication properties on page 26

## 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 REST file, set the Config property to provide the name and location of the input REST 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, specify values for any additional properties you want to configure.

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

For a connection URL:

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

### See also

HTTP header authentication properties on page 27

## 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 REST file, set the Config property to provide the name and location of the input REST 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, specify values for any additional properties you want to configure.

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

For a connection URL:

```
jdbc:subprotocol:autorest:AuthenticationMethod=UrlParameter;AuthParam=apikey;
  Config=C:/path/to/myrest.rest;SecurityToken=XaBARTsLZReM;User=jsmith;
```

### See also

URL parameter authentication properties on page 30

## 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 locations 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

OAuth 2.0 properties on page 28

## Access token flow

The access token authentication flow passes the access token directly from the client to the REST service for authentication. Unlike other grant types, authentication credentials, such as authorization codes, are not exchanged in return for the access code. Instead, the access token has been obtained from sources external to the flow and specified using the AccessToken property.

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.
- Configure the minimum required properties required for a connection:
  - If you are using a REST file, set the Config property to provide the name and location of the input REST file. For example, `C:/path/to/yelp.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`.

The following examples demonstrate a basic Yelp session using an access token flow:

Using a connection URL:

```
jdbc:subprotocol:autorest:AccessToken='C3TQH9zjwek4CgJCU-4Mxb2DxLNfI2LB3a-dNfpWYx';
   AuthenticationMethod=OAuth2;Config=C:/path/to/yelp.rest;
```

### See also

OAuth 2.0 properties on page 28

## Authorization code grant

The authorization code grant is a commonly used authentication 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 REST file, set the Config property to provide the name and location of the input REST 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`.
- 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. For example, `https://example.com/oauth2/authorize/`.
- 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, 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 basic session for a Box account using an authorization code grant:

Using a connection URL:

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

### See also

OAuth 2.0 properties on page 28

## 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 REST file, set the Config property to provide the name and location of the input REST file. For example, `C:/path/to/googleanalytics.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`.
- 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. For example, `https://example.com/oauth2/authorize/`.
- Optionally, 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 basic Google Analytics session using a client credentials grant:

Using a connection string:

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

### See also

OAuth 2.0 properties on page 28

## 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 REST file, set the Config property to provide the name and location of the input REST file. For example, `C:/path/zendesk.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`.
- 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. For example, `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, 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 basic Zendesk session using a password grant:

Using a connection string:

```
jdbc:subprotocol:autorest:AuthenticationMethod=OAuth2;
   Config=C:/path/zendesk.rest;TokenURI=https://accounts.google.com/o/oauth2/token;
   User='jjones@example.com';Password='secretstuff';
```

### See also

OAuth 2.0 properties on page 28

## 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.

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 REST file, set the Config property to provide the name and location of the input REST file. For example, `C:/path/to/googleanalytics.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`.
- 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. For example, `https://example.com/oauth2/authorize/`.
- Optionally, 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 basic Google Analytics session using a refresh token grant:

Using a connection URL:

```
jdbc:subprotocol:autorest:AuthenticationMethod=OAuth2;
    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;
```

### See also

OAuth 2.0 properties on page 28

## 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 input REST file.

**Before you start**: Define your custom authentication requests in the input REST 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 input REST 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 input REST file.
- Set the CustomAuthParams property to specify the list of parameters to be used by the authentication requests defined in the input REST file. Note that these values must be specified in the order that corresponds to the index location cited by the variable in the REST 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 REST file with custom authentication enabled:

For a connection URL:

```
jdbc:subprotocol:autorest:Config=C:/path/to/myrest.rest;CustomAuthParams=123XYZ456abc789;

       My Company Inc;path/to/endpoint;ServerName=https://example.com;User=jsmith;
       Password=secret"
```

### See also

Custom authentication requests on page 124
Custom authentication request properties on page 31

## Data encryption

The driver supports Secure Sockets Layer (SSL) data encryption. SSL works by allowing the client and server to send each other encrypted data that only they can decrypt. SSL negotiates the terms of the encryption in a sequence of events known as the *SSL handshake*. The handshake involves the following types of authentication:

- *SSL server authentication* requires the server to authenticate itself to the client.
- *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 SSL encryption

SSL encryption is enabled when the URL specified by the ServerName property file uses HTTPS. The following steps outline how to configure SSL encryption.

---

**Note:** Connection hangs can occur when the driver is configured for SSL and the database server does not support SSL. You may want to set a login timeout using the LoginTimeout property to avoid problems when connecting to a server that does not support SSL.

---

**To configure SSL encryption:**

1. Use the CryptoProtocolVersion property to specify acceptable cryptographic protocol versions (for example, TLSv1.2) supported by your server.
2. 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).
3. To validate certificates sent by the database server, set the ValidateServerCertificate property to `true`.
4. Optionally, 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.
5. If your database server is configured for SSL client authentication, configure your keystore information:
   a) 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).
   b) If any key entry in the keystore file is password-protected, set the KeyPassword property to the key password.

### See also

ServerName on page 103
CryptoProtocolVersion on page 75
TrustStore on page 109
TrustStorePassword on page 110
ValidateServerCertificate on page 111
KeyStore on page 84
KeyStorePassword on page 85
KeyPassword on page 84

## Configuring 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 SSL sockets created in the JVM.

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

  ```
  jdbc:subprotocol:autorest:Config=C:/path/to/myrest.rest;
       Password=secr3t;TrustStore=C:\Certficates\MyTruststore.jks;
       TrustStorePassword=MyTruststorePassword;User=jsmith;
  ```

  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 SSL client authentication

If the server is configured for SSL client authentication, the server asks the client to verify its identity after the server has proved its identity. Similar to 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 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`).

## IP addresses

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

The endpoint specified in the connection URL, data source, or REST file can resolve to an IPv4 or IPv6 address.

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

**Note:** You can use the Sample property or the REST file to access endpoints, for example, `https://example.com/countries/`.

Alternately, you can specify addresses using IPv4 or IPv6 format using the connection URL by using the Sample property of the or the REST file specified by the Config property. For example, the following connection URL specifies an endpoint using an IPv4 address:

```
jdbc:subprotocol:autorest:Sample='123.456.78.90';
```

You also can specify addresses in either format using a data source by using the Sample property or the REST file specified by the Config property. The following example shows a data source definition that specifies the server name using IPv6 format:

```
AutoRESTDataSource mds = new AutoRESTDataSource();
   mds.setDescription("My Autonomous REST DataSource");
   mds.setSample("[ABCD:EF01:2345:6789:ABCD:EF01:2345:6789]");
   ...
```

**Note:** When specifying IPv6 addresses in a connection URL or data source property, the address must be enclosed by brackets.

In addition to the normal IPv6 format, the drivers support IPv6 alternative formats for compressed and IPv4/IPv6 combination addresses. For example, the following connection URL specifies the server using IPv6 format, but uses the compressed syntax for strings of zero bits:

```
jdbc:subprotocol:autorest:Sample=[2001:DB8:0:0:8:800:200C:417A];
```

Similarly, the following connection URL specifies the server using a combination of IPv4 and IPv6:

```
jdbc:subprotocol:autorest:Sample=[0000:0000:0000:0000:0000:FFFF:123.456.78.90];
```

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

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

### See also

Sample on page 99
Config on page 73

## Timeouts

The driver supports the LoginTimeout, WSTimeout, and WSRetryCount connection properties. The LoginTimeout property specifies the amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request. With WSTimeout, you can specify the time, in seconds, that the driver waits for a response to a Web service request. The WSTimeout property can be used in conjunction with the WSRetryCount property. The WSRetryCount connection property can be used to retry select queries that have timed out.

**Session timeouts**

Session timeouts If the driver receives a session timeout error from a data source, the driver automatically attempts to re-establish a new session. The driver uses the initial server name, port (if appropriate), remote user ID, and remote password (encrypted) to re-establish the session. If the attempt fails, the driver returns an error indicating that the session timed out and the attempt to re-establish the session failed.

**Web service request timeouts**

You can configure the driver to never time out while waiting for a response to a Web service request or to wait for a specified interval before timing out by setting the WSTimeout connection property for fetch requests. Additionally, in a case where requests might fail, you can configure the driver to retry the request a specified number of times by setting the WSRetryCount connection property. If all subsequent attempts to retry a request fail, the driver will return an error indicating that the service request timed out and that the subsequent requests failed.

### See also

Timeout properties on page 36

## Using Java logging

The driver provides a flexible and comprehensive logging mechanism that allows logging to be incorporated seamlessly with the logging of your own application or allows logging to be enabled and configured independently from the application. The logging mechanism can be instrumental in investigating and diagnosing issues. It also provides valuable insight into the type and number of operations requested by the application from the driver and requested by the driver from the remote data source. This information can help you tune and optimize your application.

## Logging components

The driver uses the Java Logging API to configure the loggers (individual logging components) used by the driver. The Java Logging API is built into the JVM.

The Java Logging API allows applications or components to define one or more named loggers. Messages written to the loggers can be given different levels of importance. For example, errors that occur in the driver can be written to a logger at the `CONFIG` level, while progress or flow information may be written to a logger at the `FINE` or `FINER` level. Each logger used by the driver can be configured independently. The configuration for a logger includes what level of log messages are written, the location to which they are written, and the format of the log message.

The Java Logging API defines the following levels:

- `SEVERE`

- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST`

**Note:** Log messages logged by the driver only use the `CONFIG`, `FINE`, `FINER`, and `FINEST` logging levels.

Setting the log threshold of a logger to a particular level causes the logger to write log messages of that level and higher to the log. For example, if the threshold is set to `FINE`, the logger writes messages of levels `FINE`, `CONFIG`, `INFO`, `WARNING`, and `SEVERE` to its log. Messages of level `FINER` or `FINEST` are not written to the log.

The driver exposes loggers for the following functional areas:

- JDBC API
- SQL Engine
- Web service adapter

## JDBC API logger

### Name

`com.ddtek.jdbc.cloud.level`

### Purpose

Logs the JDBC calls made by the application to the driver and the responses from the driver back to the application. DataDirect Spy is used to log the JDBC calls.

### Message Levels

`FINER` - Calls to the JDBC methods are logged at the `FINER` level. The value of all input parameters passed to these methods and the return values passed from them are also logged, except that input parameter or result data contained in `InputStream`, `Reader`, `Blob`, or `Clob` objects are not written at this level.

`FINEST` - In addition to the same information logged by the `FINER` level, input parameter values and return values contained in `InputStream`, `Reader`, `Blob` and `Clob` objects are written at this level.

`OFF` - Calls to the JDBC methods are not logged.

## SQL Engine logger

### Name

`com.ddtek.cloud.sql.level`

#### Purpose

Logs the operations that the SQL engine performs while executing a query. Operations include preparing a statement to be executed, executing the statement, and fetching the data, if needed. These are internal operations that do not necessarily directly correlate with Web service calls made to the remote data source.

#### Message Levels

`CONFIG` - Any errors or warnings detected by the SQL engine are written at this level.

`FINE` - In addition to the same information logged by the `CONFIG` level, SQL engine operations are logged at this level. In particular, the SQL statement that is being executed is written at this level.

`FINER` - In addition to the same information logged by the `CONFIG` and `FINE` levels, data sent or received in the process of performing an operation is written at this level.

### Web Service Adapter logger

#### Name

`com.ddtek.cloud.adapter.level`

#### Purpose

Logs the Web service calls the driver makes to the remote data source and the responses it receives from the remote data source.

#### Message Levels

`CONFIG` - Any errors or warnings detected by the Web service adapter are written at this level.

`FINE` - In addition to the same information logged by the `CONFIG` level, information about Web service calls made by the Web service adapter and responses received by the Web service adapter are written at this level. In particular, the Web service calls made to execute the query and the calls to fetch or send the data are logged. The log entries for the calls to execute the query include the API-specific query being executed. The actual data sent or fetched is not written at this level.

`FINER` - In addition to the same information logged by the `CONFIG` and `FINE` levels, this level provides additional information.

`FINEST` - In addition to the same information logged by the `CONFIG`, `FINE`, and `FINER` levels, data associated with the Web service calls made by the Web service adapter is written.

## Configuring logging

You can configure logging using a standard Java properties file in either of the following ways:

- Using the properties file that is shipped with your JVM. See "Using the JVM" for details.
- Using the driver. See "Using the driver" for details.

### Using the JVM for logging

If you want to configure logging using the properties file that is shipped with your JVM, use a text editor to modify the properties file in your JVM. Typically, this file is named `logging.properties` and is located in the `JRE/lib` subdirectory of your JVM. The JRE looks for this file when it is loading.

You can also specify which properties file to use by setting the java.util.logging.config.file system property. At a command prompt, enter:

```
java -Djava.util.logging.config.file=properties_file
```

where:

*`properties_file`*

is the name of the properties file you want to load.

## Using the driver for logging

If you want to configure logging using the driver, you can use either of the following approaches:

- Use a single properties file for all REST sessions.
- Use a different properties file for each schema map. For example, if you have two maps (for example, `C:\data\schemamaps1\` and `C:\data\schemamaps2\`), you can load one properties file for the `test1map.config` schema map and load another properties file for the `test2map.config` schema map.

**Note:** See "SchemaMap" for information on SchemaMap default values and how to specify valid values for SchemaMap.

By default, the driver looks for the file named `ddlogging.properties` in the current working directory to load for all REST connections.

If a properties file is specified for the LogConfigFile connection property, the driver uses the following process to determine which file to load.

1. The driver looks for the file specified by the LogConfigFile property.
2. If the driver cannot find the file in Step 1 on page 57, it looks for a properties file named *`user_name`*`.logging.properties` in the directory containing the schema map for the connection, where *`user_name`* is your user ID used to connect to the REST endpoint.
3. If the driver cannot find the file in Step 2 on page 57, it looks for a properties file named `ddlogging.properties` in the current working directory.
4. If the driver cannot find the file in Step 3 on page 57 , it abandons its attempt to load a properties file.

If any of these files exist, but the logging initialization fails for some reason while using that file, the driver writes a warning to the standard output (`System.out`), specifying the name of the properties file being used.

A sample properties file is installed in the `install_dir/testforjdbc`.

### See also

SchemaMap on page 100

LogConfigFile on page 86

## Enabling Debug Record Mode

The Autonomous REST Connector supports a debug record mode that provides a method for troubleshooting issues that occur when accessing data on a REST service. When Debug Record Mode is enabled, the driver captures and records server requests and responses to a set of files stored in a designated location. Technical Support can then use these files to analyze and reproduce the issue without requiring access to your private data source.

**To generate debug record files:**

1. Using the DebugRecord property, specify the location where the driver will generate the files used to record server requests and responses.
2. Start the JDBC application and reproduce the issue.
3. Stop the application.

The driver generates a set of files containing the server requests and responses that occurred during the session. After generating the debug files, you can remove the location specified for the DebugRecord property. If you do not remove this value, the driver will overwrite debug files in the specified location the next time you start the application.

Contact Technical Support for assistance analyzing the files and reproducing the issue.

**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.

### See also

DebugRecord on page 77

## Tracking JDBC calls with DataDirect Spy

DataDirect Spy is functionality that is built into the drivers. It is used to log detailed information about calls your driver makes and provide information you can use for troubleshooting. DataDirect Spy provides the following advantages:

- Logging is JDBC 4.0-compliant.
- All parameters and function results for JDBC calls can be logged.
- Logging can be enabled without changing the application.

When you enable DataDirect Spy for a connection, you can customize logging by setting one or multiple options for DataDirect Spy. For example, you may want to direct logging to a local file on your machine.

Once logging is enabled for a connection, you can turn it on and off at runtime using the `setEnableLogging` method in the `com.ddtek.jdbc.extensions.ExtLogControl` interface.

Refer to Troubleshooting your application in the *Progress DataDirect for JDBC Drivers Reference* for information about using a DataDirect Spy log for troubleshooting.

## Enabling DataDirect Spy

You can enable and customize DataDirect Spy logging by specifying the SpyAttributes connection property for connections using a connection URL.

### Using a connection URL

The SpyAttributes connection property allows you to specify a semi-colon separated list of DataDirect Spy attributes. The format for the value of the SpyAttributes property is:

```
(
spy_attribute
[;
spy_attribute
]...)
```

where *spy_attribute* is any valid DataDirect Spy attribute. See "DataDirect Spy attributes" for a list of supported attributes.

#### Windows Example

```
jdbc:subprotocol:autorest:Config=C:path\\to\\myrest.rest;
   SpyAttributes=(log=(filePrefix)C:\\temp\\spy_;linelimit=80;logTName=yes;
   timestamp=yes)
```

---

**Note:** 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=(filePrefix)C:\\temp\\spy_`.

---

DataDirect Spy loads the driver and logs all JDBC activity to the `spy_x.log` file located in the `C:\temp` directory (`log=(filePrefix)C:\\temp\\spy_`), where *x* is an integer that increments by 1 for each connection on which the prefix is specified. The `spy_x.log` file logs a maximum of 80 characters on each line (`linelimit=80`) and includes the name of the current thread (`logTName=yes`) with a timestamp on each line in the log (`timestamp=yes`).

#### UNIX Example

```
jdbc:subprotocol:Config=~/path/to/myrest.rest;
   SpyAttributes=(log=(filePrefix)/tmp/spy_;logTName=yes;timestamp=yes)
```

DataDirect Spy loads the driver and logs all JDBC activity to the `spy_x.log` file located in the `/tmp` directory (`log=(filePrefix)/tmp/spy_`), where *x* is an integer that increments by 1 for each connection on which the prefix is specified. The `spy_x.log` file includes the name of the current thread (`logTName=yes`) with a timestamp on each line in the log (`timestamp=yes`).

#### See also

DataDirect Spy attributes on page 59

### DataDirect Spy attributes

DataDirect Spy supports the attributes described in the following table.

| Attribute | Description |
|---|---|
| `linelimit=`*`numberofchars`* | Sets the maximum number of characters that DataDirect Spy logs on a single line.<br>The default is 0 (no maximum limit). |
| `load=`*`classname`* | Loads the driver specified by *`classname`*. |
| `log=(file)`*`filename`* | Directs logging to the file specified by *`filename`*.<br>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`<br>where:<br>*X* is an integer that increments by 1 for each connection on which the prefix is specified.<br>For example, if the attribute `log=(filePrefix)C:\\temp\\spy_` is specified on multiple connections, the following logs are created:<br>`C:\temp\spy_1.log`<br>`C:\temp\spy_2.log`<br>`C:\temp\spy_3.log`<br>`...`<br>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.<br>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.<br>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.<br>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.<br>The default is no. |
| `timestamp={yes\|no}` | Specifies whether a timestamp is included on each line of the DataDirect Spy log.<br>The default is `yes`. |

3

# Connection property descriptions

You can use connection properties to customize the driver for your environment. This section lists the connection properties supported by the driver and describes each property. In the connection URL, property is expressed as a key value pair and takes the form `property=value`.

or a JDBC `DataSource`. For a connection URL

The following table provides a summary of the connection properties supported by the driver and their default values.

**Table 19: Autonomous REST Connector Properties**

| Property | Default |
|---|---|
| AccessToken on page 68 | None |
| AuthenticationMethod on page 69 | `None`<br>**Note:** The driver defaults to `Custom` when it discovers an entry for a custom authentication request in the input REST file. |
| AuthHeader on page 70 | `Authorization` |
| AuthParam on page 70 | None |
| ClientID on page 71 | None |
| ClientSecret on page 72 | None |

| Property | Default |
|---|---|
| Config on page 73 | None |
| ConvertNull on page 74 | `1` (data type check is performed if column value is null) |
| CreateMap on page 74 | `session` |
| CryptoProtocolVersion on page 75 | None |
| CustomAuthParams on page 76 | None |
| DebugRecord on page 77 | None |
| EncryptionMethod on page 78 | `noEncryption` |
| FetchSize on page 79 | `100` (rows) |
| HostNameInCertificate on page 80 | Empty string |
| ImportStatementPool on page 81 | Empty string |
| InsensitiveResultSetBufferSize on page 82 | `2048` (KB of memory) |
| KeyPassword on page 84 | None |
| KeyStore on page 84 | None |
| KeyStorePassword on page 85 | None |
| LogConfigFile on page 86 | `ddlogging.properties` |
| LoginTimeout on page 87 | `0` (no timeout) |
| LogoffURI on page 87 | None |
| MaxPooledStatements on page 88 | `0` (driver's internal prepared statement pooling is not enabled) |
| OAuthCode on page 89 | None |
| Password on page 90 | None |
| PortNumber on page 91 | When SSL encryption is enabled:<br>`80`<br>When SSL encryption is disabled:<br>`443` |
| ProxyHost on page 91 | Empty string |

| Property | Default |
|---|---|
| ProxyPassword on page 92 | Empty string |
| ProxyPort on page 93 | `0` |
| ProxyUser on page 93 | Empty string |
| ReadAhead  on page 94 | `0` |
| RedirectURI on page 95 | None |
| RefreshDirtyCache on page 96 | `1` |
| RefreshSchema on page 96 | `true` |
| RegisterStatementPoolMonitorMBean on page 98 | `false` |
| RefreshToken on page 97 | None |
| Sample on page 99 | None |
| SchemaMap on page 100 | For Windows:<br>`<application_data_folder>\ Local\Progress\DataDirect \AutoREST_Schema\`<br>For UNIX/Linux:<br>`~/progress/datadirect/ AutoREST_Schema/` |
| Scope on page 101 | None |
| SecurityToken on page 102 | None |
| ServerName on page 103 | None |
| SpyAttributes on page 103 | None |
| StmtCallLimit on page 106 | `0` (no limit) |
| StmtCallLimitBehavior on page 107 | `errorAlways` |
| Table on page 107 | None |
| TokenURI on page 108 | None |
| TrustStore on page 109 | None |
| TrustStorePassword on page 110 | None |
| User on page 110 | None |

| Property | Default |
|---|---|
| ValidateServerCertificate on page 111 | `true` |
| WSFetchSize on page 112 | `0` (maximum number of rows as set by the maximumPageSize property. The default maximum is 10,000 rows.) |
| WSPoolSize on page 113 | `1` |
| WSRetryCount on page 114 | `5` |
| WSTimeout on page 115 | `120` (seconds) |

For details, see the following topics:

- AccessToken
- AuthenticationMethod
- AuthHeader
- AuthParam
- ClientID
- ClientSecret
- Config
- ConvertNull
- CreateMap
- CryptoProtocolVersion
- CustomAuthParams
- DebugRecord
- EncryptionMethod
- FetchSize
- HostNameInCertificate
- ImportStatementPool
- InsensitiveResultSetBufferSize
- JDBCBehavior
- KeyPassword
- KeyStore
- KeyStorePassword
- LogConfigFile

- LoginTimeout
- LogoffURI
- MaxPooledStatements
- OAuthCode
- Password
- PortNumber
- ProxyHost
- ProxyPassword
- ProxyPort
- ProxyUser
- ReadAhead
- RedirectURI
- RefreshDirtyCache
- RefreshSchema
- RefreshToken
- RegisterStatementPoolMonitorMBean
- Sample
- SchemaMap
- Scope
- SecurityToken
- ServerName
- SpyAttributes
- StmtCallLimit
- StmtCallLimitBehavior
- Table
- TokenURI
- TrustStore
- TrustStorePassword
- User
- ValidateServerCertificate
- WSFetchSize
- WSPoolSize
- WSRetryCount

- WSTimeout

## AccessToken

### Purpose

Specifies the access token required to authenticate to a REST service when OAuth 2.0 is enabled (`AuthenticationMethod=OAuth2`). 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 are temporary and must be replaced to maintain the session without interruption. The life of an access token is typically one hour.
- This property is not required for all authentication flows. See "OAuth 2.0 authentication" for a list of common authentication flows and their requirements. If you are unsure of the requirements for your authentication flow, contact your administrator for more information.

### Data Source Method

`setAccessToken`

### Default

None

### Data Type

String

### See Also

- OAuth 2.0 authentication on page 44
- OAuth 2.0 properties on page 28

# AuthenticationMethod

## Purpose

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

## Valid values

`None` | `Basic` | `Custom` | `HttpHeader` | `OAuth2` | `UrlParameter`

## Behavior

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

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 `Custom`, the driver uses a custom token-based authentication flow that is defined in the input REST file. See "Custom authentication requests" for more information on the input REST file.

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 `OAuth2`, the driver uses OAuth 2.0 to authenticate to REST endpoints.

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

## Data Source Method

`setAuthenticationMethod`

## Default

`None`

**Note:** The driver defaults to `Custom` when it discovers an entry for a custom authentication request in the input REST file.

## Data Type

String

## See Also

- Custom authentication requests on page 124
- Authentication on page 42
- Basic authentication properties on page 26
- HTTP header authentication properties on page 27
- OAuth 2.0 properties on page 28
- Custom authentication request properties on page 31

- URL parameter authentication properties on page 30

## AuthHeader

### Purpose

Specifies the name of the HTTP header used for authentication. This property is used when Basic (`AuthenticationMethod=Basic`) or Header-based token authentication (`AuthenticationMethod=HttpHeader`) 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 Method

`setAuthHeader`

### Default

`Authorization`

### Data Type

String

### See Also

- AuthenticationMethod on page 69

## 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`.

### Data Source Method

`setAuthParam`

### Default

None

### Data Type

String

### See Also

- AuthenticationMethod on page 69

## ClientID

### Purpose

Specifies the client ID key for your application. The driver uses this value for certain flows when authenticating to a REST service using OAuth 2.0 (`AuthenticationMethod=OAuth2`).

### Valid Values

*`string`*

where:

*`string`*

is the client ID key for your application. For some systems, 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.

### Notes

- This property is not required for all authentication flows. See “OAuth 2.0 authentication” for a list of common authentication flows and their requirements. If you are unsure of the requirements for your authentication flow, contact your administrator for more information.

### Data Source Method

`setClientID`

### Default

None

### Data Type

String

### See also

- OAuth 2.0 authentication on page 44
- OAuth 2.0 properties on page 28

## ClientSecret

### Purpose

Specifies the client secret for your application when authenticating to a REST service with OAuth 2.0 enabled (`AuthenticationMethod=OAuth2`).

**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.

### Valid Values

*`string`*

where:

*`string`*

is the client secret for your application.

### Notes

- This property is not required for all authentication flows. See “OAuth 2.0 authentication” for a list of common authentication flows and their requirements. If you are unsure of the requirements for your authentication flow, contact your administrator for more information.

### Data Source Method

`setClientSecret`

### Default

None

### Data Type

String

### See also

- OAuth 2.0 authentication on page 44
- OAuth 2.0 properties on page 28

# Config

## Purpose

Specifies the name and location of the input REST file used to define your endpoints for sampling. This file allows you to specify multiple endpoints, define PUSH requests, and configure paging. You will need to create and specify an input REST 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 an input REST file."

## Valid Values

*`string`*

where:

*`string`*

> is the name and location of your input REST file. For example, `C:\path\to\myrest.rest` (Windows) or *`home_dir`*`/path/to/myrest.rest` (UNIX/Linux).

## Notes

- The Config property determines whether the driver uses an input REST file for the session. If no value is provided, an endpoint must be provided using the Sample property.

## Data Source Method

`setConfig`

## Default

None

## Data Type

String

## See Also

- Creating an input REST file on page 20
- Sample on page 99
- Required properties on page 24

## ConvertNull

### Purpose

Controls how data conversions are handled for null values.

### Valid Values

`0` | `1`

### Behavior

If set to `0`, 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.

If set to `1`, 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.

### Data Source Method

`setConvertNull`

### Default

`1`

### Data Type

int

### See Also

Additional properties on page 38

## CreateMap

### Purpose

Determines whether the driver creates the internal files required for a relational map of the native data when establishing a connection.

### Valid Values

`session` | `forceNew` | `notExist`

### Behavior

If set `session`, the driver uses memory to store the internal configuration information and relational map of native data. A REST file is not created when this value is specified. After the session, the view is discarded.

If set to `forceNew`, the driver deletes the current REST file, internal configuration files, and relational map in the location specified by the SchemaMap property and creates a new set at the same location.

**Warning:** This causes all map customizations defined in the REST file in the location specified by the schema map property to be lost.

If set to `notExist`, the driver uses the current REST file, internal files, and relational map in the location specified by the SchemaMap property. If the files do not exist, the driver creates them.

### Notes

Alternatively, you can refresh a schema manually at any time by using the Refresh Map statement. See "Refresh Map (EXT)" for details.

### Data Source Method

`setCreateMap`

### Default

`session`

### Data Type

String

### See Also

- SchemaMap on page 100
- Refresh Map (EXT) on page 151

## CryptoProtocolVersion

### Purpose

Specifies a cryptographic protocol or comma-separated list of cryptographic protocols that can be used when SSL is enabled (`EncryptionMethod=SSL`).

### Valid Values

`*cryptographic_protocol* [[, *cryptographic_protocol* ]...]`

where:

*`cryptographic_protocol`*

    is one of the following cryptographic protocols:

    TLSv1.2 | TLSv1.1 | TLSv1 | SSLv3 | SSLv2

**Caution:** To avoid vulnerabilities associated with SSLv3 and SSLv2, good security practices recommend using TLSv1 or higher.

### Example

If your server supports TLSv1.1 and TLSv1.2, you can specify acceptable cryptographic protocols with the following key-value pair:

```
CryptoProtocolVersion=TLSv1.1,TLSv1.2
```

### 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.
- When no value has been specified for CryptoProtocolVersion, the cryptographic protocol used depends on the highest protocol version supported by the server and the highest protocol version supported by the JDK. The driver uses the lower version of these two protocols to establish the SSL connection. Refer to the REST service documentation for information on which cryptographic protocols are supported.

### Data Source Method

setCryptoProtocolVersion

### Default

None

### Data Type

String

### See Also

- EncryptionMethod on page 78
- Data encryption on page 50

## CustomAuthParams

### Purpose

Specifies a list of parameter values used by custom authentication requests that are defined in the input REST file. This property allows you to configure parameter values used in custom authentication requests on a per connection basis, without editing the REST file, and securely pass them in a connection string or data source definition.

The input REST 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.

See "Custom authentication requests" for a detailed description of defining custom authentication requests in the input REST file.

### Valid Values

`*authentication_parameter* [[; *authentication_parameter* ]...]`

where:

*authentication_parameter*

is an authentication parameter value used in a custom authentication requests defined in the input REST 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 REST 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 Method

setCustomAuthParams

### Default

None

### Data Type

String

### See also

Custom authentication on page 49
AuthenticationMethod on page 69
Custom authentication requests on page 124

## 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 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 debeg record files are to be generated. For example, `C:\Temp\MyDebug Folder`.

### Notes

- You must have write access to the specified directory.
- For more information, refer to "Enabling Debug Record Mode" in the *Progress DataDirect for JDBC Drivers Reference*.
- For assistance, contact Technical Support.

### Data Source Method

setDebugRecord

### Default

None

### See also

Enabling Debug Record Mode on page 58

## 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 `noEncryption`, data is not encrypted or decrypted.

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.

### Notes

- SSL encryption is enabled when the URL specified in the Sample property or REST file uses HTTPS, regardless of the setting of EncryptionMethod.
- Connection hangs can occur when the driver is configured for SSL and the endpoint does not support SSL. You may want to set a login timeout using the LoginTimeout property to avoid problems when connecting to an endpoint that does not support SSL.
- 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 Source Method

setEncryptionMethod

### Default

`noEncryption`

### Data Type

String

### See Also

Data encryption on page 50

Performance considerations on page 41

## 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 Method

setFetchSize

### Default

`100` (rows)

### Data Type

Int

### See Also

- Additional properties on page 38
- Performance considerations on page 41

## HostNameInCertificate

### Purpose

Specifies a host name for certificate validation when SSL encryption is enabled (`EncryptionMethod=SSL`) 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 a DNSName value does not exist in the SubjectAlternativeName or if the certificate does not have a SubjectAlternativeName, the driver compares the host name with the Common Name (CN) part of the certificate's Subject name. If the values do not match, the connection fails and the driver throws an exception.

### Notes

- If the HostNameInCertificate is not specified, the driver automatically uses the value of the ServerName from the URL as the value for validating the certificate.
- 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 method

`setHostNameInCertificate`

### Default

Empty string

### Data type

String

### See also

- EncryptionMethod on page 78
- ValidateServerCertificate on page 111

## 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 Method

`setImportStatementPool`

### Default

empty string

### 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 Method

`setInsensitiveResultSetBufferSize`

### Default

`2048`

### Data Type

int

### See also

- Additional properties on page 38
- Performance considerations on page 41

## JDBCBehavior

### Purpose

Determines how the driver describes native 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 JVM. This allows your application to continue using JDBC 3.0 types.

### Data Source Method

`setJDBCBehavior`

### Default

`1`

### Data Type

int

### See Also

- Data type handling properties on page 35

## KeyPassword

### Purpose

Specifies the password that is used to access the individual keys in the keystore file when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the REST server. This property is useful when individual keys in the keystore file have a different password than the keystore file.

### Valid Values

*`string`*

where:

*`string`*

is a valid password.

### Data Source Method

`setKeyPassword`

### Default

None

### Data Type

String

### See Also

- EncryptionMethod on page 78
- Data encryption properties on page 32

## KeyStore

### Purpose

Specifies the directory of the keystore file to be used when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the REST server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.

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.

### Valid Values

*`string`*

where:

*`string`*

is a valid directory of a keystore file.

### Notes

- The keystore and truststore files can be the same file.

### Data Source Method

`setKeyStore`

### Default

None

### Data Type

String

### See Also

- EncryptionMethod on page 78
- Data encryption properties on page 32

## KeyStorePassword

### Purpose

Specifies the password that is used to access the keystore file when SSL is enabled (`EncryptionMethod=SSL`) and SSL client authentication is enabled on the REST server. The keystore file contains the certificates that the client sends to the server in response to the server's certificate request.

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.

### Valid Values

*`string`*

where:

*`string`*

is a valid password.

### Notes

- The keystore and truststore files can be the same file.

### Data Source Method

setKeyStorePassword

### Default

None

### Data Type

String

### See Also

- EncryptionMethod on page 78
- Data encryption properties on page 32

## 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.

### Data Source Method

`setLogConfigFile`

### Default

`ddlogging.properties`

### Data Type

String

### See Also

Refer to "Using Java logging" in the *Progress DataDirect for JDBC Drivers Reference*.

# LoginTimeout

### Purpose

The amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request.

### Valid Values

`0` | *x*

where:

*x*

is a positive integer that represents a number of seconds.

### Behavior

If set to `0`, the driver does not time out a connection request.

If set to *x*, the driver waits for the specified number of seconds before returning control to the application and throwing a timeout exception.

### Data Source Method

`setLoginTimeout`

### Default

`0`

### Data Type

int

### See Also

- Timeout properties on page 36

# 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 Method

`setLogoffURI`

### Default

None

### Data Type

String

### See Also

- OAuth 2.0 authentication on page 44
- OAuth 2.0 properties on page 28

## MaxPooledStatements

### Purpose

The maximum number of pooled prepared statements for this connection. Setting MaxStatements to an integer greater than zero (0) enables the driver's internal prepared statement pooling, which is useful when the driver is not running from within an application server or another application that provides its own prepared statement pooling.

### Valid Values

`0` | *x*

where

*x*

is a positive integer that represents a number of pooled prepared statements.

### Behavior

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

If set to *x*, the driver enables the DataDirect Statement Pool and uses the specified value to cache a certain number of 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.

### Data Source Method

`setMaxPooledStatements`

### Default

`0`

### Data Type

int

### See Also

- Statement pooling properties on page 37

## OAuthCode

### Purpose

Specifies the temporary authorization code that is exchanged for access tokens when OAuth 2.0 authentication is enabled (`AuthenticationMethod=OAuth2`). Authorization codes are used to authenticate against the endpoint specified by the 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.

### Valid Values

*`string`*

where:

*`string`*

is an OAuth 2.0 authorization code.

### Notes

- This property is not required for all authentication flows. See “OAuth 2.0 authentication” for a list of common authentication flows and their requirements. If you are unsure of the requirements for your authentication flow, contact your administrator for more information.

### Data Source Method

`setAuthCode`

### Default

None

### Data Type

String

### See Also

- OAuth 2.0 authentication on page 44
- OAuth 2.0 properties on page 28

## Password

### Description

Specifies the password to use to connect to your REST service. This property is ignored when `AuthenticationMethod=None`.

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

### Valid Values

*`password`*

where:

*`password`*

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

### Data Source Method

`setPassword`

### Default

None

### Data Type

String

### See Also

- AuthenticationMethod on page 69
- Basic authentication properties on page 26

## PortNumber

### Purpose

Specifies the TCP port of the server that is listening for REST API requests.

### Valid Values

*`port`*

where:

*`port`*

is the port number.

### Data Source Method

`setPortNumber`

### Default

When SSL encryption is disabled:

`80`

When SSL encryption is enabled:

`443`

### Data Type

int

### See Also

- Required properties on page 24

## ProxyHost

### Description

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. See "Using IP addresses" for details about using these formats.

### Data Source Method

`setProxyHost`

### Default

Empty string

### See also

IP addresses on page 53
Connecting through a proxy server on page 40
Proxy server properties on page 33

## 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 Method

`setProxyPassword`

### Default

Empty string

### See Also

- Connecting through a proxy server on page 40
- Proxy server properties on page 33

# 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 Method

`setProxyPort`

## Default

`0`

## See Also

- Connecting through a proxy server on page 40
- Proxy server properties on page 33

# 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 Method

`setProxyUser`

### Default

Empty string

### See Also

- Connecting through a proxy server on page 40
- Proxy server properties on page 33

## 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.

### 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 Method

`setReadAhead`

### Default

`0`

### Data Type

int

### See Also

- Additional properties on page 38

## RedirectURI

### Purpose

Specifies the endpoint to which the client is returned after authenticating with a third-party service when OAuth 2.0 authentication is enabled (`AuthenticationMethod=OAuth2`).

For some authentication flows, the REST service will redirect you to a third-party service for authentication. Once your credentials have been validated, the third-party service returns the client to an endpoint in the REST service to continue the session. The endpoint used by the third-party service is provided by the client and specified using the RedirectURI property.

### Valid Values

*`string`*

where:

*`string`*

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

### Notes

- The redirect endpoint is often registered with the authentication service to provide improved security. Registering the endpoint prevents your valid authentication credentials being redirected to a malicious site; therefore, reducing the risk of sharing your access token and other sensitive information with unauthorized parties.
- This property is not required for all authentication flows. See "OAuth 2.0 authentication" for a list of common authentication flows and their requirements. If you are unsure of the requirements for your authentication flow, contact your administrator for more information.

### Data Source Method

`setRedirectURI`

### Default

None

### Data Type

String

### See Also

- OAuth 2.0 authentication on page 44
- OAuth 2.0 properties on page 28

## 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

`1 | 0`

### Behavior

If set to `1`, a dirty cache is refreshed when the cache is referenced in a fetch operation. The cache state is set to initialized if the refresh succeeds.

If set to `0`, a dirty cache is not refreshed when the cache is referenced in a fetch operation.

### Data Source Method

`setRefreshDirtyCache`

### Default

`1`

### Data Type

Int

## RefreshSchema

### Purpose

Specifies whether the driver automatically refreshes the map of the data model when a user connects to a REST service.

### Valid Values

`true | false`

### Behavior

If set to `true`, the driver automatically refreshes the map of the data model when a user connects to a REST service. Changes to objects since the last time the map was generated will be shown in the metadata.

If set to `false`, the driver does not refresh the map of the data model when a user connects to a REST service.

### Notes

- This property should not be enabled (`RefreshSchema=true`) when `CreateMap=session`.

### Data Source Method

`setRefreshSchema`

### Default

`true`

### See Also

- Mapping properties on page 25

## RefreshToken

### Purpose

Specifies the refresh token used to either request a new access token or renew an expired access token. The value for this property is used for certain flows when authenticating to a REST service with OAuth 2.0 enabled (`AuthenticationMethod=OAuth2`).

**Important:** The refresh token is a confidential value used to authenticate to the server. To prevent unauthorized access, this value must be securely maintained.

### Valid Values

*`string`*

where:

*`string`*

is the refresh token you have obtained from the authentication service.

### Notes

- This property is not required for all authentication flows. See “OAuth 2.0 authentication” for a list of common authentication flows and their requirements. If you are unsure of the requirements for your authentication flow, contact your administrator for more information.

### Data Source Method

`setRefreshToken`

### Default

None

### Data Type

String

### See Also

- OAuth 2.0 authentication on page 44
- OAuth 2.0 properties on page 28

## 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.

### Data Source Method

setRegisterStatementPoolMonitorMBean

### Default

`false`

### Data Type

Boolean

### See also

Statement pooling properties on page 37
MaxPooledStatements on page 88

# 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 an input REST file. Note that if your session does any of the following, instead of using this property, you must create an input REST 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

*`string`*

where:

*`string`*

is the endpoint that you want connect to and sample. For example, `https://example.com/countries/`.

## 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.

## Data Source Method

`setSample`

## Default

None

## Data Type

String

## See Also

- Config on page 73
- Table on page 107
- Required properties on page 24

## SchemaMap

### Purpose

Specifies the directory where the internal configuration files, REST file, and the relational map of the REST data model are written. The driver looks for these files when connecting to a REST service. If the file does not exist, the driver creates one.

### Valid Values

*`string`*

where:

*`string`*

is the path to the directory used to store the configuration files, REST file, and relational map. For example, if SchemaMap is set to a value of `C:\\Users\\Default\\AppData\\Local\\Progress\\DataDirect\\AutoREST_Schema\\`, the driver either creates or looks for these files in the directory `C:\Users\Default\AppData\Local\Progress\DataDirect\AutoREST_Schema`.

### Notes

- By default, the name of the internal files are determined by the values of the User and ServerName connection properties. If no value is specified for either property, a prefix of `USER` is used for these files. For example, `USER.config`.
- When connecting to a REST service, the driver looks for the schema map configuration files in the specified location. If the configuration files do not exist, the driver creates them using the location you have provided. If you do not provide a location, the driver creates it using default values.
- You can refresh the internal files related to an existing view of your data by using the SQL extension Refresh Map. Refresh Map runs a discovery against your native data and updates your internal files accordingly.

### Example

As the following examples show, escapes are needed when specifying SchemaMap for a data source but are not used when specifying SchemaMap in a connection URL.

#### Connection URL Example

```
jdbc:datadirect:autorest:Config=C:\\path\\to\\myrest.rest;
SchemaMap=C:\\Users\\Default\\AppData\\Local\\Progress\\DataDirect\\AutoREST_Schema\\
```

#### Data Source Example

```
AutoRESTDataSource ds = new AutoRESTDataSource();
ds.setDescription("My Autonomous REST Connector Data Source");
ds.setConfig("C:\\path\\to\\myrest.rest");
ds.setSchemaMap("C:\\Users\\Default\\AppData\\Local\\Progress
              \\DataDirect\\AutoREST_Schema\\")
```

### Data Source Method

`setSchemaMap`

### Default

- For Windows XP and Windows Server 2003
    - `user_profile\Application Data\Local\Progress\DataDirect\AutoREST_Schema\`
- For other Windows platforms
    - User data source: `user_profile\AppData\Local\Progress\DataDirect\AutoREST_Schema\`
    - System data source: `C:\Users\Default\AppData\Local\Progress\DataDirect\AutoREST_Schema\`
- For UNIX/Linux
    - `home_dir/progress/datadirect/AutoREST_Schema/`

### Data Type

String

### See Also

- Refresh Map (EXT) on page 151
- Mapping properties on page 25

## Scope

### Purpose

Specifies a space-separated list of OAuth scopes that limit the permissions granted by an access token. The driver uses this value when authenticating to a REST service using OAuth 2.0 (`AuthenticationMethod=OAuth2`).

### Valid Values

*`string`*

where:

*`string`*

is a space-separated list of security scopes.

### Examples

The following example demonstrates a configuration that limits the user to read only access to user permissions and data using Google Analytics API.

```
Scope=https://www.googleapis.com/auth/analytics.manage.users.readonly
      https://www.googleapis.com/auth/analytics.readonly
```

### Data Source Method

`setScope`

### Default

None

### Data Type

String

### See Also

- OAuth 2.0 authentication on page 44
- OAuth 2.0 properties on page 28

## SecurityToken

### Purpose

Specifies the security token required to make a connection to your REST API endpoint. This property is required when token based authentication is enabled (`AuthenticationMethod=HttpHeader | UrlParameter`); otherwise, this property is ignored. 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.

---

**Important:** If setting the security token using a data source, be aware that the SecurityToken property, like all data source properties, is persisted in clear text.

---

### Valid Values

*`string`*

where:

*`string`*

is the value of the security token assigned to the user.

### Data Source Method

`setSecurityToken`

### Default

None

### Data Type

String

### See Also

- AuthenticationMethod on page 69
- Basic authentication properties on page 26

## ServerName

### Purpose

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

### Valid Values

`string`

where:

`string`

is the host name portion of the URL endpoint to which you send requests. For example, `https://example.com`.

### Data Source Method

`setServerName`

### Default

None

### Data Type

String

### See Also

- Required properties on page 24

## SpyAttributes

### Purpose

Enables DataDirect Spy to log detailed information about calls 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.<br>The default is 0 (no maximum limit). |
| `load=`*`classname`* | Loads the driver specified by *`classname`*. |
| `log=(file)`*`filename`* | Directs logging to the file specified by *`filename`*.<br>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`<br>where:<br>*X* is an integer that increments by 1 for each connection on which the prefix is specified.<br>For example, if the attribute `log=(filePrefix)C:\\temp\\spy_` is specified on multiple connections, the following logs are created:<br>`C:\temp\spy_1.log`<br>`C:\temp\spy_2.log`<br>`C:\temp\spy_3.log`<br>`...`<br>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.<br>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.<br>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.<br>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.<br>The default is no. |
| `timestamp={yes\|no}` | Specifies whether a timestamp is included on each line of the DataDirect Spy log.<br>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`.
- For more information, refer to "Tracking JDBC calls with DataDirect Spy" in the *Progress DataDirect for JDBC Drivers Reference*.

## Data Source Method

`setSpyAttributes`

## Default

None

# 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 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 Method

`setStmtCallLimit`

## Default

`0` (no limit)

## Data Type

int

## See Also

- Web service properties on page 34

## StmtCallLimitBehavior

### Purpose

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

### Valid Values

`errorAlways` | `returnResults`

### Behavior

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

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.

### Data Source Method

`setStmtCallLimitBehavior`

### Default

`errorAlways`

### Data Type

String

### See Also

- Web service properties on page 34

## 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 an input REST 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 Method

`setSample`

### Default

None

### Data Type

String

### See Also

- Config on page 73
- Required properties on page 24

## TokenURI

### Purpose

Specifies the endpoint used to exchange authentication credentials for access tokens when OAuth 2.0 authentication is enabled (`AuthenticationMethod=OAuth2`). The credentials passed to this endpoint depend on the authentication flow being employed by your REST service. For a list of credentials required for common authentication flows, refer "OAuth 2.0 authentication."

### Valid Values

*`string`*

where:

*`string`*

is the endpoint used to retrieve OAuth 2.0 access tokens. For example:

`https://example.com/oauth2/authorize/`

### Notes

- This property is not required for all authentication flows. See “OAuth 2.0 authentication” for a list of common authentication flows and their requirements. If you are unsure of the requirements for your authentication flow, contact your administrator for more information.

### Data Source Method

`setTokenURI`

### Default

None

### Data Type

String

### See Also

- OAuth 2.0 authentication on page 44
- OAuth 2.0 properties on page 28

## TrustStore

### Purpose

Specifies the directory of the truststore file to be used when SSL is enabled (`EncryptionMethod=ssl`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

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 `ValidateServerCertificate=false`.

### Valid Values

*`string`*

where:

*`string`*

is the directory of the truststore file.

### Data Source Method

`setTrustStore`

### Default

None

### Data Type

String

### See Also

- EncryptionMethod on page 78
- ValidateServerCertificate on page 111
- Performance considerations on page 41

- Data encryption properties on page 32

## TrustStorePassword

### Purpose

Specifies the password that is used to access the truststore file when SSL is enabled (`EncryptionMethod=SSL`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

This value overrides the password of the truststore file that is specified by the javax.net.ssl.trustStorePassword Java system property. If this property is not specified, the truststore password is specified by the javax.net.ssl.trustStorePassword Java system property.

This property is ignored if `ValidateServerCertificate=false`.

### Valid Values

*`string`*

where:

*`string`*

is a valid password for the truststore file.

### Data Source Method

`setTrustStorePassword`

### Default

None

### Data Type

String

### See Also

- EncryptionMethod on page 78
- ValidateServerCertificate on page 111
- Performance considerations on page 41
- Data encryption properties on page 32

## User

### Purpose

Specifies the user name that is used to connect to the REST service. A user name is required if user is enabled by your REST service. This property is ignored when `AuthenticationMethod=None`.

### Valid Values

*`string`*

where:

`string`

is a valid user name. The user name is case-insensitive.

### Data Source Method

`setUser`

### Default

None

### Data Type

String

### See Also

- Basic authentication properties on page 26

## ValidateServerCertificate

### Purpose

Determines whether the driver validates the certificate that is sent by the REST service server when SSL encryption is enabled (`EncryptionMethod=SSL`). When using SSL server authentication, any certificate that is sent by the server must be issued by a trusted Certificate Authority (CA).

Allowing the driver to trust any certificate that is 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`

### Behavior

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 method

`setValidateServerCertificate`

### Default

`true`

### Data type

Boolean

### See also

- EncryptionMethod on page 78
- HostNameInCertificate on page 80

## 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, specifying paging parameters for an endpoint in the input REST file. See "Creating an input REST 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 input REST file.

### Behavior

If set to `0`, the driver attempts to fetch up to the maximum number of row specified by the maximumPageSize property. This value typically provides the maximum throughput.

If set to *x*, the driver attempts to fetch up to the 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 WSFetch size 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 Method

setWSFetchSize

### Default

`0` (up to the maximum number of rows specified by the maximumPageSize property)

### Data Type

Int

### See Also

- Creating an input REST file on page 20
- Web service properties on page 34
- Performance considerations on page 41

## 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 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 Method

`setWSPoolSize`

### Default

`1`

### Data Type

Int

### See also

- Web service properties on page 34
- Performance considerations on page 41

## WSRetryCount

### Description

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 Method

setWSRetryCount

### Default

5

### Data Type

Int

### See Also

- Web service properties on page 34

# 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 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 Method

setWSTimeout

## Default

`120` (seconds)

## Data Type

Int

## See Also

- Web service properties on page 34

4

# Input REST file syntax

The driver employs an input REST file to map JSON responses to the relational model. Although the primary purpose of the REST 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 REST file.

The input REST 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. The following is the basic structure of the REST file:

```
{
   "#http":[<http_response_codes>],

   "#<oauth2_param>":"<oauth2_value>",

   "#authentication":[<custom_auth>],
   "#reauthentication":[<custom_auth>],

   "<table_name1>":"<table_definition1>",
   "<table_name2>":"<table_definition2>",
   "<table_name3>":"<table_definition3>"
}
```

**Note:** With the exception of table definitions, all REST entries described in the following table are optional.

**Table 20: Input REST file components**

| Lines | Entry/Entry Type | Description |
|---|---|---|
| 2 | `#http` | Defines how HTTP response status codes are processed by the driver.<br><br>For details and syntax, see HTTP response code processing on page 119. |
| 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.<br><br>For details and syntax, see OAuth 2.0 authentication on page 122. |
| 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.<br><br>For details and syntax, see Custom authentication requests on page 124. |
| 7 | `#reauthentication` | Defines the request used to refresh the access token retrieved through the `#authentication` entry.<br><br>For details and syntax, see Custom authentication requests on page 124. |
| 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.<br><br>For details and syntax, see Table definition entries on page 126. |

For details, see the following topics:

- HTTP response code processing
- OAuth 2.0 authentication
- Custom authentication requests
- Table definition entries
- Example input REST file

## HTTP response code processing

The driver allows you to customize how HTTP response status codes are processed by the driver. This provides you with a method to define error responses for codes that are returned by the service, including subsequent driver actions and error messages. By using the `#operation` and `#match` properties, you can further limit the definition to apply only to responses for certain operations or for service responses that contain specific content, such as a specific error message.

If no `#http` entry is created to define how response codes are processed, the driver handles response codes according to the default actions in the following table. Note that if you supply your own `#http` entry, you must provide a definition for each response code that the service returns. Any code not explicitly defined in your entry will be returned as a failure.

| Code | Action |
|---|---|
| 200 | OK |
| 400 | ZERO_ROWS |
| 401 | REAUTHENTICATE |
| 404 | ZERO_ROWS |
| 429 | RETRY_AFTER |
| 503 | RETRY_AFTER |

An entry to define HTTP response code processing takes the following form for multiple definitions:

```
"#http":[
    {
        "#code":<code_number1>,
        "#action":"<action>",
        "#operation":"<operation>",
        "#match":"<match_string>",
        "#message":"<message>"
    },
    {
        "#code":<code_number2>,
        "#action":"<action>",
        "#operation":"<operation>",
        "#match":"<match_string>",
        "#message":"<message>"
    },
    {
        "#code":<code_number3>,
        "#action":"<action>",
        "#operation":"<operation>",
        "#match":"<match_string>",
        "#message":"<message>"
    }
]
```

**Important:** The driver reads definitions in the order listed and uses the first one that matches the response being evaluated. Therefore, if you have multiple definitions for a single code response, it's important to specify definitions with `#match` and `#operation` parameters before definitions without conditions. This helps the driver avoid using a definition that is designed to generally apply to a code before evaluating those with specific conditions.

*`code_number`*

is the numeric HTTP status code for which you want to define driver behavior. For example, `200`, `401`, `404`, etc.

*`action`*

is the action the driver takes after the specified HTTP status code is returned and the values of the `operation` and `match` properties are determined to apply. A list of supported actions is described in the "Action Values" table.

*`operation`*

(optional) is the operation types to which you want to limit the response definition to apply. For example, if you only wanted the behavior defined in this entry to apply to instances when performing insert operations, set this value to `Select`. See the "Operation Values" table for a list of supported values and their definition.

*`match`*

(optional) is text used to limit the instances to which the response definition applies. When a value is provided for this property, the driver scans the first 512 bytes of the service response for the specified value. If it's detected, the driver determines that the behavior in the definition applies. For example, if you only wanted the behavior to apply when the response body contained the text `"status":"error"`, you would specify the following:

```
"#match":"\"status\":\"error\"",
```

*`message`*

(optional) is the message text that you want the driver to return when encountering the specified status code. This is typically the error message you want returned to the user. You can specify this value as plain text or as a reference to a header of the server response. For example, to reference the "message" header in a service response, you would specify the following to display the text contained in the "message" header:

```
"#message":"{message}"
```

**Table 21: 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 22: 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 input REST 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 an input REST 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 REST 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 input REST 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 23: 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

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 input REST 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 REST 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 REST 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 REST 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`.

## Example: Simple token request

The following is an example of a simple token request, where `access-token` is the server response that contains the payload with the access token. Most custom authentication requests will take this form.

```
"#authentication" : [
//Header
    "api-key={CustomAuthParams[1]}",
//Payload
    {
        "credentials": {
            "username": "{user}",
            "password": "{password}",
            "company": "{customAuthParams[2]}"
        }
    },
//HTTP request
    "POST http://{serverName}/bearertoken",
//Data request credentials. "{/access-token}" refers to the service
//response from the preceding HTTP request.
    "HEADER Authentication=Bearer {/access-token}"
]
```

## Example: Two-step token request

The following example demonstrates a two-step authentication, where the service response from the initial request, `UserToken`, is passed in the request header of the second stage of authentication. The principles demonstrated in this example apply to authentication flows requiring two or more requests.

```
"#authentication" : [
//Header request for first request
    "accept=application/json",
    "content-type=application/json",
    "kmauthtoken=\\{sitename:\"{customAuthParams[1]}\",localeId:\"en_US\"\\}\\}",
//Payload for first request
    {
        "login": "{user}",
        "password": "{password}",
        "siteName": "{customAuthParams[1]}"    },
//HTTP request for first token
    "POST https://{serverName}/getusertoken",
//Header for second request. "{/authenticationToken}" refers to the value of
//the service response from the preceding HTTP request.
    "accept=application/json",
    "content-type=application/json",
    "kmauthtoken=\\{sitename:\"{CustomAuthParams[1]}\",localeId:\"en_US\",
     UserToken:\"{/authenticationToken}\"\\}",
//Payload for second request
    {
        "userName": "{user}",
        "password": "{password}",
        "siteName": "{customAuthParams[1]}",
        "userExternalType": "ACCOUNT"
    },
//HTTP request for second token
    "POST https://{serverName}/getaccesstoken",
//Data request credentials. "{/customAuthToken}" refers to the value in
//the service response from the preceding HTTP request.
    "HEADER Authentication=Bearer
        \\{sitename:\"{customAuthParams[1]}\",localeId:\"en_US\",
           userToken:\"{/authenticationToken}\",integrationUserToken:\
           "{/accessToken}\"\\}"]
```

## Table definition entries

Table definition entries define the mapping of JSON endpoints to tables. You can specify a single entry or, in a comma separated list, multiple entries. These entries can be as simple as a colon-separated table name and endpoint pair (`"<table_name>":"<endpoint>"`,). Or, for a greater level of configuration, entries can take the form of JSON objects.

The following demonstrates the syntax of a set of three simple table definition entries. For endpoint details and syntax, see "Query paths."

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

The following demonstrates the syntax used to configure a single table entry in an array.

**Note:** The following example demonstrates the syntax for all the features and functionality supported by the driver, but it is not typical for defining a table. In most scenarios, only a subset of these parameters would be used to define a table.

```
1 {
2   "<table_name>": {
3      "#path":["<endpoint>"],
4      "#<paging_parameter>":"<paging_value>",
5      "#<parsing_parameter>":"<parsing_value>",
6   // The following POST entry defines two fields.You can define one or more
7   // fields in an entry.
8      "#post":{"<field1>":"<value1>","<field2>":"<value2>"}
9   //The following HTTP header entry defines two headers. You can define one or more
10  //headers in an entry.
11     "#headers":{"<header1>":"<value1>","<header2>":"<value2>"}
12     "<column1>":"<data_type>",
13     "<column2>":"<data_type>,#key",
14  //The following array defines two nested columns. You can define one or more
15  //nested columns in an array.
16    "<column3>[]":{"<column_a>:<data_type>","<column_b>":"<data_type>"}
17  //The following key-map defines two columns. You can define one or more
18  //columns in a key map.
19    "<column4>{<data_type>}":{"<column_c>:<data_type>","<column_d>":"<data_type>"}
20  //The following column defines two nested objects. You can define one or more
21  //nested columns in table definition.
22     "<column5>":{"<nested_column1>":"<data_type>","<nested_column2>":"<data_type>"}
23     "<column6>":"<data_type>","<java_date_format>"
24     "<column7>":{"#type":"<data_type>","#extract:<reg_expression>"}
25     "<column8>":{"#type":"<data_type>","#header":true,"#eq":"<header_name>"}
26     "<column9>":{"#type":"<data_type>","#<operator>":"<uri_property>"}
27     }
28 }
```

**Table 24: Components of a table definition entry**

| Lines | Entry/Entry Type | Description |
|---|---|---|
| 2 | Table name | (Required) Specifies the name of the table. |
| 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.<br><br>For details and syntax, see Query paths on page 135. |
| 4 | 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.<br><br>For details and syntax, see Paging on page 129. |
| 5 | 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.<br><br>For details and syntax, see REST model parsing on page 131. |

| Lines | Entry/Entry Type | Description |
|---|---|---|
| 8 | `#post` | Defines the sample values used when issuing a POST request. This section is mutually exclusive with the `#headers` property.<br>For details and syntax, see POST requests on page 131. |
| 11 | `#headers` | Specifies the HTTP headers to filter data returned by a GET request. This section is mutually exclusive with the `#post` property.<br>For details and syntax, see Requests with custom HTTP headers on page 133. |
| 12-26 | 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.<br>For details and syntax, see Column names on page 138 and Data type mapping on page 139. |
| 13 | Primary key | Designates the primary key by specifying the `#key` element in a column definition.<br>For details and syntax, see Primary key on page 141. |
| 16 | Column as an array | Defines a column as an array by specifying brackets (`[]`) at the end of its column name.<br>For details and syntax, see Columns as an array on page 141. |
| 19 | Column as key-value map | Defines a column as an key-value map by specifying brackets (`{}`) at the end of its column name.<br>For details and syntax, see Columns as a key-value map on page 142. |
| 22 | Column with nested objects | Defines a column with nested objects in the entry body.<br>For details and syntax, see Columns with nested objects on page 142. |
| 23 | Time stamp formats | Defines the time stamp format for a column in the definition.<br>For details and syntax, see Date, time, and timestamp formats on page 143. |
| 24 | `#extract` | Specifies a regular expression that allows you to extract a subfield, or portion, of a string value.<br>For details and syntax, see Subfields on page 144. |
| 25 | `#header` | Specifies whether the column can be sent as an HTTP header instead of part of a query string for GET requests.<br>For details and syntax, see Columns as HTTP headers on page 144. |
| 26 | Filtering and URI parameters | Specifies filtering operations to be sent in requests for the column.<br>For details and syntax, see Filtering and URI parameters on page 145. |

### See also

Query paths on page 135

## 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 REST 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.

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>",
          "#maximumPageSize":1000,
          "#firstRowNumber":1,
          "#pageSizeParameter":"maxResults",
          "#rowOffsetParameter":"startAt"
 },
```

### Row offset paging

The following table describes the parameters used to configure row offset paging:

**Table 25: Row Offset Paging Properties**

| Property | Description |
|---|---|
| `#maximumPageSize` | Specifies the maximum page size in rows. |
| `#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 26: Page Number Paging Properties**

| Property | Description |
|---|---|
| `#maximumPageSize` | Specifies the maximum page size in rows. |
| `#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:

**Table 27: Page Number Paging Properties**

| Property | Description |
|---|---|
| `#MaximumPageSize` | (Optional) Specifies the maximum page size in rows. This option is only required when the page size is not dictated by the data source. |
| `#PageSizeParameter` | (Optional) Specifies the name of the URI parameter that contains the page size. |
| `#NextPageElement` | Specifies the name of the element containing 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 slash-separated path. |
| `#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`. |

## 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 28: 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 29: 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:<br>`{"Name":Sam,"Pet":"dog","vehicle":"car"}`<br>`{"Name":Denise,"Pet":"sugar bear","vehicle":"bike"}`<br>The default is `false`. |

## POST requests

To use POST requests, you must define the request in the REST 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.

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 query 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>"
          }
     },
```

*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.

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 an 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 query parameters.

```
"incidents": {
          "#path": "https://www.example.com/safety/",
          "#post": {
               "departments":"accounting",
               "date":"2015-10-8",
               }
           "reported":{
                  "#type":"date",
                  "#eq":"reportedOn"
          }
     },
```

## Requests with custom 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 use custom headers, you must define the request in the input REST file. The REST 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.

In addition to filtering requests, the header object can be used to specify a value for the Accept header if the default, `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 ore 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`.

*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"
      }
},
```

## 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.

*`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",
```

## Requests with unparameterized paths

Unparametrized requests are issued as GET requests, unless they are specified in a POST request entry. To specify endpoints for unparameterized requests, use the following format:

```
"<host_name>/<endpoint_path>",
```

*host_name*

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the ServerName property.

*endpoint_path*

is the path component of the URL endpoint. For example, `countries`.

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 131

## Requests with parameterized paths

Parameterized requests are issued as GET requests, unless they are specified in a POST request entry. To specify parameterized requests, use the following format:

```
"<host_name>/<endpoint_path1>/{<param_name>:<param_value>}[/<endpoint_path2>]",
```

*host_name*

(optional) is the protocol and host name components of the URL endpoint. For example, `http://example.com`. You can omit this value by specifying the host name using the ServerName property.

*endpoint_path*

is the path component of the URL endpoint. For example, `states`.

*param_name*

is the parameter identifier used for filtering the request. For example, `countryCode`.

*param_value*

is the parameter value used for filtering the request during sampling. For example, `USA`.

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

```
#path:"http://example.com/states/get/{countryCode:USA}/all",
```

### See also

POST requests on page 131

## 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`.

*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

POST requests on page 131

## 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 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
```

## 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 30: 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 | Range: `-99*10`$^{8}$ to `99*10`$^{8}$ |
| Decimal(*p.s*) | Range: `-99*10`$^{14}$ to `99*10`$^{14}$<br>Minimum scale: `0`<br>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*) | Precision: `12`<br>Minimum scale: `0`<br>Maximum scale: `9` |
| Timestamp(s) | Precision: `23`<br>Minimum scale: `0`<br>Maximum scale: `9` |
| TinyInt | Range: `-999` to `999` |
| VarBinary(*l*) | Precision: `16777215` |

| Data Type and Parameters | Characteristics |
|---|---|
| VarChar(*l*) | Precision: 32767 |

## Primary key

You can designate the primary key for a table by modifying the REST file. In the column object, add the `#key` after the data type element, separated by a comma. 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",
           }
}
```

## 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 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. The syntax to send a column as an HTTP header takes the following form:

```
"<column_name>[]":{"#type":"<data_type>","#header":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 REST 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 REST 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.

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 31: Query operator syntax**

| Query Operator | Property syntax |
|---|---|
| `=` | `"#eq":"<uri_property>"` |

| Query Operator | Property syntax |
|---|---|
| `<` | `"#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-O1-1
  ```

- 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=20
  ```

- 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
  ```

## Example input REST file

The following is an example input REST 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"
      },
  }
```

5

# 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 32: 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:<br>`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:<br>`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:<br>`ALTER SESSION SET autorest.WS_CALL_COUNT=0`<br>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:<br>`SELECT * from information_schema.system_remote_sessions WHERE session_id = cursessionid()` |

# 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_clausefrom_clause
[where_clause]
[groupby_clause]
[having_clause]
[{UNION [ALL | DISTINCT] |
 {MINUS [DISTINCT] | EXCEPT [DISTINCT]} |
 INTERSECT [DISTINCT]} select_statement]
[limit_clause]
```

where:

*select_clause*

specifies the columns from which results are to be returned by the query. See "Select" for a complete explanation.

*from_clause*

specifies one or more tables on which the other clauses in the query operate. See "From" for a complete explanation.

*where_clause*

is optional and restricts the results that are returned by the query. See "Where clause" for a complete explanation.

*`groupby_clause`*

is optional and allows query results to be aggregated in terms of groups. See "Group By clause" for a complete explanation.

*`having_clause`*

is optional and specifies conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). See "Having clause" for a complete explanation.

`UNION`

is an optional operator that combines the results of the left and right Select statements into a single result. See "Union operator" for a complete explanation.

`INTERSECT`

is an optional operator that returns a single result by keeping any distinct values from the results of the left and right Select statements. See "Intersect operator" for a complete explanation.

`EXCEPT | MINUS`

are synonymous optional operators that returns a single result by taking the results of the left Select statement and removing the results of the right Select statement. See "Except and Minus operators" for a complete explanation.

*`orderby_clause`*

is optional and sorts the results that are returned by the query. See "Order By clause" for a complete explanation.

*`limit_clause`*

is optional and places an upper bound on the number of rows returned in the result. See "Limit clause" for a complete explanation.

## Select clause

### Purpose

Use the Select clause to specify with a list of column expressions that identify columns of values that you want to retrieve or an asterisk (*) to retrieve the value of all columns.

### Syntax

```
SELECT [{LIMIT offsetnumber | TOP number}] [ALL | DISTINCT] {* | column_expression
[[AS] column_alias] [,column_expression [[AS] column_alias], ...]}
```

where:

`LIMIT` *`offset number`*

creates the result set for the Select statement first and then discards the first number of rows specified by *`offset`* and returns the number of remaining rows specified by *`number`*. To not discard any of the rows, specify `0` for *`offset`*, for example, `LIMIT 0` *`number`*. To discard the first *`offset`* number of rows and return all the remaining rows, specify `0` for *`number`*, for example, `LIMIT` *`offset`*`0`.

`TOP` *`number`*

is equivalent to `LIMIT 0`*`number`*.

*`column_expression`*

can be simply a column name (for example, `last_name`). More complex expressions may include mathematical operations or string manipulation (for example, `salary * 1.05`). See "SQL expressions" for details. *`column_expression`* can also include aggregate functions. See "Aggregate functions" for details.

*`column_alias`*

can be used to give the column a descriptive name. For example, to assign the alias department to the column dep:

```
SELECT dep AS department FROM emp
```

`DISTINCT`

eliminates duplicate rows from the result of a query. This operator can precede the first column expression. For example:

```
SELECT DISTINCT dep FROM emp
```

### Notes

- Separate multiple column expressions with commas (for example, `SELECT last_name, first_name, hire_date`).
- Column names can be prefixed with the table name or table alias. For example, `SELECT emp.last_name or e.last_name`, where `e` is the alias for the table `emp`.
- NULL values are not treated as distinct from each other. The default behavior is that all result rows be returned, which can be made explicit with the keyword `ALL`.

### See also

SQL expressions on page 161

## 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 33: 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 169

## 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 169
SQL expressions on page 161

## 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 169

SQL expressions on page 161

## 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 169

SQL expressions on page 161

## 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 161

## 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
```

# 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 34: Literal Syntax Examples**

| SQL Type | Literal Syntax | Example |
|---|---|---|
| BIGINT | *n*<br>where<br>*n* is any valid integer value in the range of the INTEGER data type | `12 or -34 or 0` |
| BOOLEAN | Min Value: 0<br>Max Value: 1 | `0`<br>`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`<br>where:<br>*n*<br>is the integral part<br>*f*<br>is the fractional part | `0.25`<br>`3.1415`<br>`-7.48` |
| DOUBLE | `n.fEx`<br>where:<br>*n* is the integral part<br>*f* is the fractional part<br>*x* is the exponent | `1.2E0` or `2.5E40` or `-3.45E2` or `5.67E-4` |
| INTEGER | *n*<br>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 35: 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 36: 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 37: 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<br><br>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.<br><br>The default escape character is backslash (\). | `SELECT * FROM emp WHERE ENAME LIKE 'J%\_%' ESCAPE '\'`<br><br>This matches all records with names that start with letter 'J' and have the '_' character in them.<br><br>`SELECT * FROM emp WHERE ENAME LIKE 'JOE\_JOHN' ESCAPE '\'`<br><br>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 38: 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 39: 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 40: Conditions**

| Condition | Description |
|---|---|
| Simple comparison | Specifies a comparison with expressions or subquery results.<br>`= , !=, <>, < , >, <=, <=` |
| Group comparison | Specifies a comparison with any or all members in a list or subquery.<br>`[= , !=, <>, < , >, <=, <=] [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)
```