Snowflake Plug-In User Guide

Macedon Technologies

The following user guide presents information on how to configure and use the Snowflake plug-in. It provides steps on setting up the connected system and related integrations, as well as detailing and providing examples on the various pieces of functionality provided by the plug-in.

macedontechnologies.com
info@macedontechnologies.com

Table of Contents

1. Plug-In

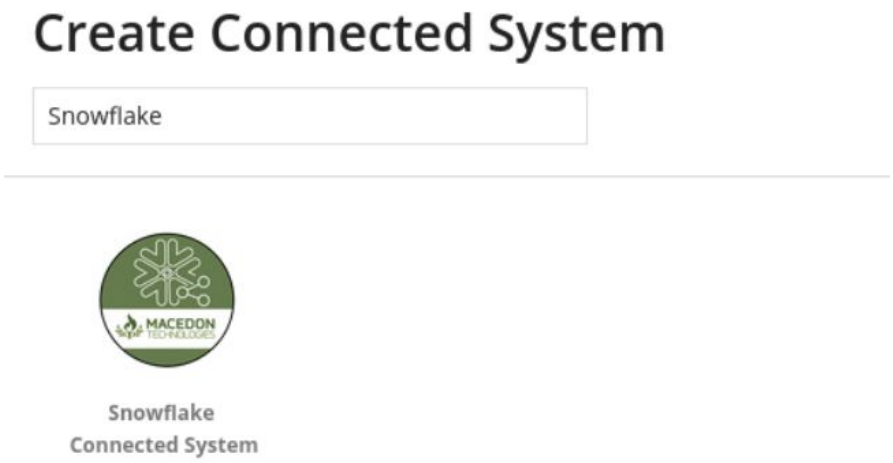
The functionality created for interfacing with Snowflake is packaged as a deployable plug-in for an Appian environment. It should be added to the directory *<APPIAN_HOME>/_admin/plugins* after which it will be installed by Appian when the directory is scanned for plugins.

2. Connected System

The Snowflake plug-in exposes a connected system that can be configured in Appian. The connected system allows for connecting to Snowflake database instances that can be leveraged in addition to the OOTB Appian MySQL instance.

2.1 Creation

To create a new Connected System to connect to your Snowflake instance in Appian, the 'Snowflake Connected System' must be selected.



While creating the Connected System, enter the connection information including 'Account Name', 'User Name', 'Password','Role', 'Warehouse', 'Database', and 'Schema' for which you would like this Connected System to connect to a Snowflake instance.

Clicking the 'Test Connection' button will allow you to verify the connection before proceeding. Be sure to save the connection details for this connected system.

Connected System Properties

Snowflake Connected System

Connect with a Snowflake database to execute SELECT, INSERT, UPDATE, DELETE statements or execute a stored procedure or user defined function.
Version: 2

Name *

CS_SnowFlakeV2

Description

Connect to a Snowflake account

Snowflake Connected System Configuration

Account Name *

abc12345.us-east-1

User Name *

mcdnsnowflake

Password

********** (Clear)

Role *

PUBLIC

Warehouse

TEST_WH

Database *

TEST_DB

Schema *

PUBLIC

TEST CONNECTION

CANCEL

SAVE

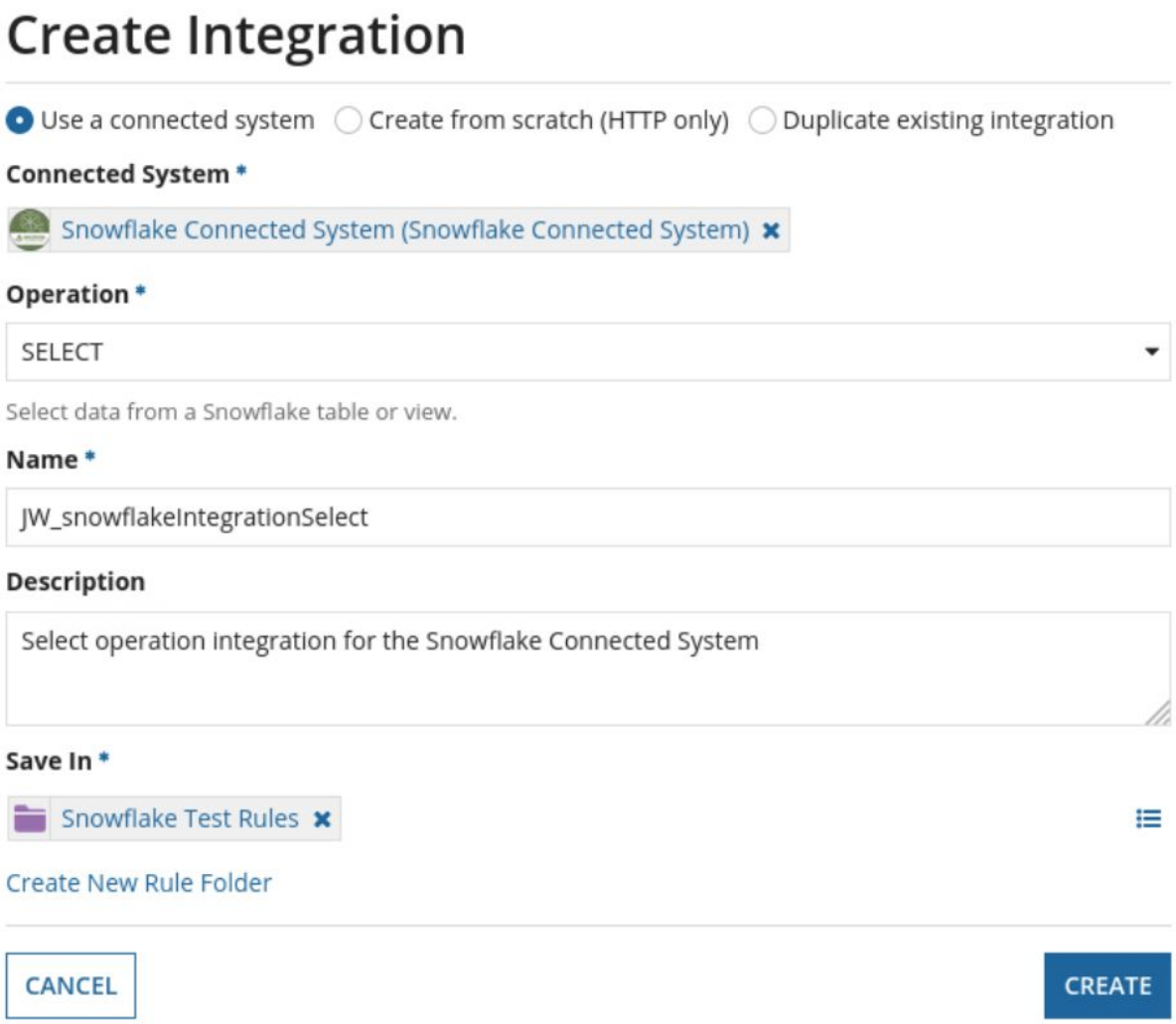
3. Integration

Six types of integrations are available for the Snowflake connected system, modeled around a typical CRUD model: Select, Insert, Update, and Delete. Version two also supports calling a user defined function or executing a stored procedure. The available integrations allow for a way to interface with a Snowflake database instance in order to view and manipulate its contents.

3.1 Configuration

For the Snowflake plugin, you can create an Integration object the same as you would for any other integration in Appian. You can do this by clicking the 'Use in New Integration' button while configuring a Connected System or by creating a new Integration object from the Appian Designer view.

Selecting an operation lets you set what type query the integration object will be able to execute. The options will be Select, Insert, Update and Delete.



3.2 Select

The Select integration allows for querying against a Snowflake instance and pulling back data. Various fields are provided which allow for customization of the columns to select, filters to apply, and paging to limit final results.

Creating an integration for making Select queries requires a Snowflake Connected System and then selecting the 'Select Query' operation upon creation.

In order to properly query a Snowflake table, the table must first be specified. After selecting a table, additional fields will be available for query creation. These fields are designed to emulate the corresponding Appian query entity components to allow for more seamless querying.

3.2.1 Query Columns

The Query Columns field determines the columns to return as part of the result set. The field is designed to emulate a query entity selection by allowing for a list of Appian dictionaries in the form of `a!queryColumn` components (`a!querySelection` is *not* supported). Listed inputs can be of type `a!queryColumn` or a dictionary.

Default Behavior (No Input)

All columns are returned

Syntax

Format 1

```
a!queryColumn(
  field: <Text>,
  alias: <Text>
)
```

Format 2

```
{
  field: <Text>,
  alias: <Text>
}
```

Note: Not all `a!queryColumn` fields are supported, as shown

Example

The results of this query would contain two columns: 'RIN' (containing the value of the 'id' column) and 'description'

```
{
  a!queryColumn(
    field: “id”,
    alias: “RIN”
  ),
  a!queryColumn(
    field: “description”
  )
}
```

3.2.2 Query Filters

The Query Filters field is for specifying the filters to apply to the query. The field is designed to emulate query entity filters by allowing for a list of Appian dictionaries in the form of `a!queryFilter` components. All given filters are combined using the `AND` operator and `a!queryLogicalExpression` is *not* supported. Listed inputs can be of type `a!queryColumn` or a dictionary.

Default Behavior (No Input)

No rows are filtered

Syntax

Format 1

```
a!queryFilter(
  field: <Text>,
  operator: <Text>,
  value: <Any Type>
)
```

Format 2

```
{
  field: <Text>,
  operator: <Text>,
  value: <Any Type>
}
```

Acceptable operators are `'='`, `'<>'`, `'>'`, `'>='`, `'<'`, `'<='`, `'between'`, `'in'`, `'not in'`, `'is null'`, `'not null'`, `'starts with'`, `'not starts with'`, `'ends with'`, `'not ends with'`, `'includes'`, and `'not includes'`

Note: Not all `a!queryFilter` fields are supported, as shown

Example

This query would return all rows that had a ‘statusId’ of 1, 2, or 3 and were marked as active

```
{
  a!queryFilter(
    field: “statusId”,
    operator: “in”,
    value: {1, 2, 3}
  ),
  a!queryFilter(
    field: “isActive”,
    operator: “=”,
    value: true
  )
}
```

3.2.3 Sort Info

The Sort Info field is for specifying the sorting criteria for the query results. The field is designed to emulate the `a!sortInfo` component, taking in a field to sort by and the direction of sorting (ascending or descending). Sort Info is its own dedicated field and not part of the Paging Info (3.2.4) field because of technical limitations with Appian connected system plug-in development. Inputs can be given using the dropdown and radio button fields, or provided as an expression of type `a!sortInfo` or a dictionary.

Default Behavior (No Input)

If no field is provided, the default Snowflake sorting is used. If a field is provided but the direction is not, the results will be sorted by the given field in descending order.

Syntax

The below structures can be followed if the field input is an expression

Format 1

```
a!sortInfo(
  field: <Text>,
  ascending: <Boolean>
)
```

Format 2

```
{
  field: <Text>,
  ascending: <Boolean>
}
```

Example

This result would be sorted by the ‘priority’ field starting with the lowest value and ending with the highest value

```
a!sortInfo(
  field: “priority”,
  ascending: true
)
```

3.2.4 Paging Info

The Paging Info field is for specifying the paging for the query results. The field is designed to emulate the `a!pagingInfo` component, taking in a batch size and starting index to retrieve a subset of the full results. Inputs can be given using the text fields, or provided as an expression of type `a!pagingInfo` or a dictionary.

Default Behavior (No Input)

All results are returned

Syntax

Format 1

```
a!pagingInfo(
  batchSize: <Number>,
  startIndex: <Number>
)
```

Format 2

```
{
  batchSize: <Number>,
  startIndex: <Number>
}
```

Note: Not all `a!pagingInfo` fields are supported, as shown

Example

This query would return the first ten rows from the complete results

```
a!pagingInfo(
  batchSize: 10,
  startIndex: 1
)
```

3.3 Insert

The Insert integration supports the ability to insert new rows in a Snowflake table.

Creating an integration for making Insert queries requires a Snowflake Connected System and then selecting the ‘Insert Query’ operation upon creation.

In order to properly insert into a Snowflake table, the database, schema, and table must first be specified. These fields will become available in order of their hierarchy (Database -> Schema -> Table). After selecting a table, an additional field will be available for query creation. This field is designed to emulate the existing way Appian handles inserting CDTs to allow for more seamless additions.

3.3.1 Values

The Values field is for providing the data to insert into the Snowflake database. The field takes in a list of CDT values corresponding to the schema of the selected Snowflake table to be inserted. Any fields not specified that are part of the table will contain the default value configured for the column in Snowflake. Listed inputs can be of the relevant CDT type or a dictionary.

Default Behavior (No Input)

No rows are inserted into the database

Syntax

Format 1

```
type!<CDT>(
  <field>: <type>,
  …
)
```

Format 2

```
{
  <field>: <type>,
  …
}
```

Example

These two rows would be inserted into the Employee table, assuming 'title' has a default value of `NULL`: (1, "Sarah Smith", "Supervisor") and (2, "Bikram Bajwa", `NULL`)

Note: A value for the primary key is not required for inserting rows into a table. If a table has auto-increment configured for its primary key, the value will be automatically populated by Snowflake. If multiple rows are being inserted at the same time, either all or none of the rows need a primary key value. If there is a mix of provided and not provided primary keys, an error will occur.

```
{
  type!Employee(
    id: 1,
    name: “Sarah Smith”,
    title: “Supervisor”
  ),
  type!Employee(
    id: 2,
    name: “Bikram Bajwa”
  )
}
```

3.4 Update

The Update integration supports the ability to update existing rows in a Snowflake table.

Creating an integration for making Update queries requires a Snowflake Connected System and then selecting the ‘Update Query’ operation upon creation.

In order to properly update rows in a Snowflake table, the table must first be specified. After selecting a table, an additional field will be available for query creation. This field is designed to emulate the existing way Appian handles updating rows to allow for more seamless additions.

3.4.1 Values

The Values field is for providing the rows to update in the Snowflake database. The field takes in a list of CDT values corresponding to the schema of the selected Snowflake table to be inserted. Any fields not specified that are part of the table will be updated to have a value of `NULL`. Since matching is performed on the primary key, a primary key must be populated for each row to update. Listed inputs can be of the relevant CDT type or a dictionary.

Note: Only integer primary keys are currently supported by the Snowflake plug-in

Default Behavior (No Input)

No rows are updated in the database

Syntax

Format 1

```
type!<CDT>(
  <field>: <type>,
  …
)
```

Format 2

```
{
  <field>: <type>,
  …
}
```

Example

The existing rows with 'id' (primary key) would be updated to (1, "Sarah Jones Smith", "Supervisor") and (2, "Bikram Singh Bajwa", NULL)

```
{
  type!Employee(
    id: 1,
    name: "Sarah Jones Smith",
    title: "Supervisor"
  ),
  type!Employee(
    id: 2,
    name: "Bikram Singh Bajwa"
  )
}
```

3.5 Delete

The Delete integration supports the ability to delete rows in a Snowflake table.

Creating an integration for making Delete queries requires a Snowflake Connected System and then selecting the 'Delete Query' operation upon creation.

In order to properly delete rows in a Snowflake table, the table must first be specified. After selecting a table, an additional field will be available for query creation. This field is designed to simplify the existing way Appian handles deleting rows to allow for easy row deletions.

3.5.1 Identifiers

The Identifiers field is for providing the list of primary keys of the rows to delete from the Snowflake database. The field takes in a list of integers corresponding to the primary key of the selected Snowflake table.

Note: Only integer identifiers are currently supported by the Snowflake plug-in

Default Behavior (No Input)

No rows are deleted from the database

Syntax

```
{<primary_key>, …}
```

Example

The rows with primary keys of '35' and '72' would be deleted

```
{35, 72}
```

3.6 Call User Defined Function

The Call Function operation supports the ability to call a user defined function and return the result set, which can be parsed just like a Select operation.

In order to call a function, the function must first be selected from the dropdown. After it is selected, a grid will be generated which displays the list of parameters and their types. The values of these parameters are expressionable.

Default Behaviour (No Input)

The function is called with null values passed as parameters.

Syntax

{<field_name_1>, ...}

Example

The function will be called passing a single parameter named LENGTH1 with the value 3.1.

{

LENGTH1: 3.14

}

3.7 Execute Stored Procedure

The Execute Stored Procedure operation exists in two versions, one for read-only operations, and one for operations that write data. The designer should choose the operation based on the behavior of the stored procedure.

In order to execute a stored procedure, the procedure must be selected from the dropdown. After it is selected, a grid is generated which lists the parameter names of the procedure and their types. The values of these parameters are expressionable.

Default Behaviour (No Input)

The procedure is called with null values passed as parameters.

Syntax

{<field_name_1>, ...}

Example

The function will be called passing a single parameter named FLOAT_PARAM1 with the value 3.1.

```
{
  FLOAT_PARAM1: 3.1
}
```

4. Permissions

In order to call and configure the integrations the plugin supports, the login account and role specified in the connected system configurations must have the necessary underlying permissions in Snowflake to carry out the operations. The role must be granted access to the underlying database and schema. In addition, when configuring integration objects, the plugin makes some calls against the information schema. Examples of these calls are below:

DESC TABLE "TB_TEST1";

DESCRIBE PROCEDURE STPROC1(FLOAT);

If any of these permissions are not granted, the integration may be impossible to configure, or the call could fail. Designers should coordinate with a Snowflake administrator to configure the necessary access.

For example, we may have a stored procedure DELETE_RECORD that takes a single varchar as a parameter, but not be able to see this present in the dropdown on an execute stored procedure integration. It order for it to show up when using a role called WAREHOUSE_CLIENT, grant the following permission in Snowflake:

grant usage on procedure DELETE_RECORD(varchar) TO ROLE WAREHOUSE_CLIENT;

Below are a basic set of grants that allow us to call SELECT/INSERT/UPDATE/DELETE operations on all tables in a schema, as well as query any view:

GRANT USAGE ON DATABASE TEST_DB TO ROLE WAREHOUSE_CLIENT;

GRANT USAGE ON SCHEMA PUBLIC TO ROLE WAREHOUSE_CLIENT;

GRANT SELECT,INSERT,UPDATE,DELETE ON ALL TABLES IN SCHEMA TEST_DB.PUBLIC TO ROLE WAREHOUSE_CLIENT;

GRANT SELECT ON ALL VIEWS IN SCHEMA TEST_DB.PUBLIC TO ROLE WAREHOUSE_CLIENT;