GoodData Foreign Data Wrapper

Release 1.3.0

GoodData Corporation

Mar 10, 2023

CONTENTS:

GoodData Foreign Data Wrapper delivers PostgreSQL foreign data wrapper extension built on top of multicorn. The extension makes GoodData.CN insights, computations and ad-hoc report data available in PostgreSQL as tables. It can be selected like any other table using the SQL language.

CHAPTER

ONE

LIMITATIONS

FDW is suitable for small to medium insights results or scheduled reports created in the third party tools. Carefully consider use in a production environment.

1.1 Installation

You can build and run it as a service in your Docker environment (recommended) or install the package on your system directly using pip.

1.1.1 Requirements

- Python 3.7 or newer
- GoodData.CN or GoodData Cloud

1.1.2 Install Using Docker (Recommended)

The Python SDK comes with a Dockerfile which, when started, will run PostgreSQL 12 with multicorn and `gooddata-fdw` pre-installed. For an even better user experience there is a `docker-compose.yaml` file which contains both the `gooddata-fdw` and `gooddata-cn-ce` services.

Build and Run the Service

Execute the following command in your repository root folder:

```
docker-compose up --build -d
```

`gooddata-fdw` image is built from the Dockerfile and both services are started in background.

Note: Services in docker-compose.yaml contain a setup of various environment variables including `POSTGRES_PASSWORD`. Feel free to set the variables in your environment, before you execute the above command. Default value for `POSTGRES_PASSWORD` is `gooddata123`.

Maintenance

To rebuild the Foreign Data Wrapper image execute the following command:

```
docker-compose build
```

If you would like to purge a container completely (including the volume) and start from scratch, run the following helper scripts:

```
./rebuild.sh gooddata-cn-ce
./rebuild.sh gooddata-fdw
```

Adding Your Own Data

Before you start playing with the Foreign Data Wrapper, you will need a content in the `gooddata-cn-ce`.

`docker-compose.yaml` launches the *upload-layout* service. Its purpose is to bootstrap the demo and testing content into gooddata-cn-ce. You can use this as a starting point.

But the `gooddata-cn-ce` service is not limited only to the demo content. You can fill the `gooddata-cn-ce` with your own content (LDM, metrics, insights). Follow our Getting Started documentation if you need help with that.

Connect with existing GoddData.CN installation

This use case is for users running a GoodData.CN image who want to connect it to the GoodData Foreign Data Wrapper. For connecting `gooddata-fdw` with GoodData.CN image both images have to run on the same network. You can create a new network and run both images there or use the default bridge network.

Note: Default network bridge does not support accessing services by their name. You need to use an IP address in the host when defining the GoodData.CN server. The IP address can be found using command `docker inspect <GoodData.CN container name>`.

---

1. Build the `gooddata-fdw` service from `docker-compose.yaml`:

   ```
   docker-compose build gooddata-fdw
   ```

2. Create a new network:

   ```
   docker network create --driver bridge gd-cn-net
   ```

3. Run GoodData.CN on created network and name it `gooddata-cn-ce`:

   ```
   docker run --rm --name gooddata-cn-ce -p 3000:3000 -p 5432:5432 -v /data \
   --network gd-cn-net \
   -e LICENSE_AND_PRIVACY_POLICY_ACCEPTED=YES \
   -e APP_LOGLEVEL=INFO \
   gooddata/gooddata-cn-ce:latest
   ```

4. Run the `gooddata-fdw` service on created network and name it `postgres-fdw`:

   ```
   docker run --rm --name postgres-fdw -p 2543:5432 --network gd-cn-net \
   -e POSTGRES_DB=gooddata -e POSTGRES_USER=gooddata -e POSTGRES_PASSWORD=gooddata123 \
   ```

(continues on next page)

(continued from previous page)

```
gooddata-python-sdk_gooddata-fdw:latest \
postgres -c "shared_preload_libraries=foreign_table_exposer" -c "log_statement=all"␣
↪-c "client_min_messages=DEBUG1" -c "log_min_messages=DEBUG1"
```

1.1.3 Install Using Pip

Run the following command to install the `gooddata-fdw` package on your system:

```
pip install gooddata-fdw
```

Warning: For this use case, you also need to install and run PostgreSQL together with multicorn.

1.2 Get Started With PostgreSQL

1.2.1 Connect to PostgreSQL

After the `gooddata-fdw` container starts, you can connect to the running PostgreSQL:

- From console using `psql --host localhost --port 2543 --user gooddata gooddata`

  You will be asked to enter the password that you have specified when starting the script.

- From any other client using JDBC string: `jdbc:postgresql://localhost:2543/gooddata`

  You will be asked to enter username (gooddata) and password.

Once connected you will be able to work with the GoodData.CN Foreign Data Wrapper. At first, you need to define your GoodData.CN server in PostgreSQL:

```
CREATE SERVER multicorn_gooddata FOREIGN DATA WRAPPER multicorn
OPTIONS (
    wrapper 'gooddata_fdw.GoodDataForeignDataWrapper',
    host 'https://gooddata-cn-ce:3000', -- host equal to name of container with GoodData.
↪CN.CE
    token 'YWRtaW46Ym9vdHN0cmFwOmFkbWluMTIz' -- default gooddata-cn-ce token, documented␣
↪in public DOC as well
);
```

As of now the GoodData.CN community edition (single container deployment) supports only `localhost` as the target host. If you spin-up GoodData.CN and FDW using docker-compose, GoodData.CN host name is the service name in the docker-compose, e.g. `gooddata-cn-ce`. To enable such setup, we provide an option `header_host`:

```
CREATE SERVER multicorn_gooddata FOREIGN DATA WRAPPER multicorn
  OPTIONS (
    wrapper 'gooddata_fdw.GoodDataForeignDataWrapper',
    host 'http://gooddata-cn-ce:3000', -- host equal to name of container with GoodData.
↪CN.CE
    token 'YWRtaW46Ym9vdHN0cmFwOmFkbWluMTIz', -- default gooddata-cn-ce token,␣
↪documented in public DOC as well
```

(continues on next page)

(continued from previous page)

```
    headers_host 'localhost'
  );
```

Typically, you have to do this once per GoodData.CN installation. You may add as many servers as you need.

IMPORTANT: Do not forget to specify host including the schema (http or https).

1.3 Foreign Tables

1.3.1 Import GoodData Objects into PostgreSQL Schema

You can import insights created in GoodData.CN Analytical Designer as PostgreSQL foreign tables. You can import insights from as many workspaces and/or GoodData.CN instances (servers) as you want.

You can also import your entire semantic model including MAQL metrics into a special `compute` *pseudo-table*. Doing SELECTs from this table will trigger computation of analytics on your GoodData.CN server based on the columns that you have specified on the SELECT.

Note: The `compute` is called pseudo-table for a reason. It does not adhere to the relational model. The columns that you SELECT map to facts, metrics and labels in your semantic model. Computing results for the select will automatically aggregate results on the columns that are mapped to labels in your semantic model. In other words cardinality of the `compute` table changes based on the columns that you SELECT.

For your convenience we prepared a stored procedure, which:

- (re)creates target schema
- imports currently existing insights and/or entire semantic model

You can re-execute the procedure to update foreign tables.

```
-- This maps all insights stored in GoodData.CN workspace `workspace_id` into the␣
↪PostgreSQL schema named `workspace_id`
CALL import_gooddata('workspace_id', 'insights');

-- By utilizing the third parameter you can override the name of the target PostgreSQL␣
↪schema
CALL import_gooddata('workspace_id', 'insights', 'custom_schema');

-- This imports the semantic model into the 'compute' pseudo-table.
CALL import_gooddata('workspace_id', 'compute');

-- This imports both insights and compute
CALL import_gooddata('workspace_id', 'all');

-- This is how you can extend max size of numeric columns in foreign tables (basically␣
↪to support larger numbers)
CALL import_gooddata(workspace := 'goodsales', object_type := 'all', numeric_max_size :=␣
↪24);

-- Specify custom foreign server name - this enables you importing from multiple servers␣
↪into the same FDW instance
```

(continues on next page)

(continued from previous page)

```
CALL import_gooddata(workspace := 'goodsales', object_type := 'all', foreign_server :=
↪'multicorn_gooddata_stg');
```

Default max numeric size is 18, default digits after decimal point is 2 unless metric format defines more.

You will get a couple of 'NOTICE' messages as the import progresses. You can then check the imported tables by executing:

```
SELECT * FROM information_schema.foreign_tables WHERE foreign_table_schema = 'workspace_
↪id';
```

IMPORTANT: Your semantic model may consist of multiple isolated segments that have no relationship between them. Attempting to compute results from multiple isolated segments will result in errors.

> **Warning:** Imported tables reflect state of the workspace and insights in time of import. Any later change to the workspace can result in failing SQL queries against imported tables. The state can be fixed by re-importing the workspace insights and/or compute.

1.3.2 Create Foreign Tables

You can manually create your own foreign tables and map their columns to GoodData.CN semantic model. This is similar to creating normal tables except you have to provide table and column OPTIONS to establish the correct mapping. For instance:

```
CREATE FOREIGN TABLE custom_report (
    some_label VARCHAR OPTIONS (id 'label/some_label'),
    some_fact_sum  NUMERIC(15,5) OPTIONS (id 'fact/some_fact', agg 'sum'),
    some_fact_avg  NUMERIC(15,5) OPTIONS (id 'fact/some_fact', agg 'avg'),
    some_metric  NUMERIC(15,5) OPTIONS (id 'metric/some_metric')
)
SERVER multicorn_gooddata
OPTIONS ( workspace 'workspace_id');
```

To explain:

- OPTIONS on foreign table must contain identifier of workspace to map to
- OPTIONS on each column must contain identifier of semantic model entity. The id is string but consisting of two parts `<entity_type>/<entity_id>`. Where `entity_type` is either label, fact or metric.

For columns that map to facts in your semantic model, you can also specify what aggregation function should be used when aggregating the fact values for the labels in your custom report table. You can use the following aggregation functions:

- `sum`
- `avg`
- `min`
- `max`
- `median`

The `agg` key is optional. If you do not specify it, then default `sum` aggregation will be used. The value of `agg` is case insensitive.

Note: If you do not specify the required options, the CREATE command will fail. If you specify wrong entity IDs, the failures will happen at SELECT time.

1.3.3 Push Down of Filters

When querying foreign tables, you can add `WHERE` clause filtering the result. For performance optimization, it makes sense to push such filters down to the GoodData.CN, so not all data has to be collected.

We are able to push only some filters down to GoodData.CN:

- Simple attribute(label) filters
  - Example: `WHERE region IN ('East', 'West')`
- Simple date filters
  - Only DAY granularity is supported
  - (NOT) IN operator is `not` supported
  - Example: `WHERE my_date BETWEEN '2021-01-01 AND 2021-02-01`

If you use an `OR` between conditions, it is not pushed down. Push down is possible in case of custom tables and `compute` table, not in case of foreign tables imported from `insights`.

1.3.4 Known Limitations

It is not possible to reference a column in `WHERE` clause, which is not used in `SELECT` section. Example:

```
SELECT label1, metric FROM insight WHERE label2 = 'a';
SELECT label1, metric FROM compute WHERE label2 = 'a';
```

While it is obvious in case of an `insight` (it does not contain the column at all), in case of `compute` we would like to support it, but we are not allowed due to lack of functionality in Multicorn - the filter is always applied on final result set and if it does not contain the column, it does not work.

1.4 API

gooddata_fdw

1.4.1 gooddata_fdw

Modules

gooddata_fdw.column_utils	
gooddata_fdw.column_validation	
gooddata_fdw.environment	This file exists because multicorn is not available as proper stand-alone python package that one could install and then use the different data types during testing.
gooddata_fdw.executor	
gooddata_fdw.fdw	
gooddata_fdw.filter	
gooddata_fdw.import_workspace	
gooddata_fdw.naming	
gooddata_fdw.options	
gooddata_fdw.pg_logging	
gooddata_fdw.result_reader	

gooddata_fdw.column_utils

Functions

column_data_type_for(attribute)	Determine what postgres type should be used for *attribute*.
table_col_as_computable(col)	

gooddata_fdw.column_utils.column_data_type_for

gooddata_fdw.column_utils.**column_data_type_for**(*attribute: Optional[CatalogAttribute]*) → str

Determine what postgres type should be used for *attribute*.

Parameters

attribute – catalog attribute instance

gooddata_fdw.column_utils.table_col_as_computable

gooddata_fdw.column_utils.**table_col_as_computable**(*col:* ColumnDefinitionStub) → Union[Attribute, Metric]

gooddata_fdw.column_validation

Functions

validate_columns_in_table_def(table_columns, ...)

gooddata_fdw.column_validation.validate_columns_in_table_def

gooddata_fdw.column_validation.**validate_columns_in_table_def**(*table_columns: dict[str, ColumnDefinition]*, *query_columns: list[str]*) → None

Classes

ColumnValidator()
IdOptionValidator(mandatory)
LocalIdOptionValidator()

gooddata_fdw.column_validation.ColumnValidator

class gooddata_fdw.column_validation.**ColumnValidator**

Bases: object

__init__()

Methods

__init__()
validate(column_name, column_def)

gooddata_fdw.column_validation.IdOptionValidator

class gooddata_fdw.column_validation.**IdOptionValidator**(*mandatory: bool*)

Bases: *ColumnValidator*

__init__(*mandatory: bool*)

Methods

__init__(mandatory)
validate(column_name, column_def)

gooddata_fdw.column_validation.LocalIdOptionValidator

class gooddata_fdw.column_validation.**LocalIdOptionValidator**

Bases: *ColumnValidator*

__init__()

Methods

__init__()
validate(column_name, column_def)

gooddata_fdw.environment

This file exists because multicorn is not available as proper stand-alone python package that one could install and then use the different data types during testing.

The multicorn python code is part of the PostgreSQL extension installation.

Thus here is the layer of indirection that tries to import multicorn code and if that is not present (likely during test run) it will use stub implementations.

The stubbing only happens if the FDW code is called during test execution. Otherwise the import error is raised as usual to prevent some wicked behavior on mis-configured PostgreSQL.

Functions

log_to_postgres(msg, level)	

gooddata_fdw.environment.log_to_postgres

gooddata_fdw.environment.**log_to_postgres**(*msg: str*, *level: int*) → None

Classes

ColumnDefinition	alias of *ColumnDefinitionStub*
ColumnDefinitionStub(column_name, type_name, ...)	
ForeignDataWrapper	alias of *ForeignDataWrapperStub*
ForeignDataWrapperStub(options, columns)	
Qual	alias of *QualStub*
QualStub(field_name, operator, value)	
TableDefinition	alias of *TableDefinitionStub*
TableDefinitionStub(table_name, columns, options)	

gooddata_fdw.environment.ColumnDefinition

gooddata_fdw.environment.**ColumnDefinition**

alias of *ColumnDefinitionStub*

gooddata_fdw.environment.ColumnDefinitionStub

class gooddata_fdw.environment.**ColumnDefinitionStub**(*column_name: str*, *type_name: str*, *options: dict[str, str]*)

Bases: `object`

__init__(*column_name: str*, *type_name: str*, *options: dict[str, str]*) → None

Methods

__init__(column_name, type_name, options)	

gooddata_fdw.environment.ForeignDataWrapper

gooddata_fdw.environment.**ForeignDataWrapper**

alias of *ForeignDataWrapperStub*

gooddata_fdw.environment.ForeignDataWrapperStub

class gooddata_fdw.environment.**ForeignDataWrapperStub**(*options: dict[str, str]*, *columns: dict[str, ColumnDefinition]*)

Bases: object

__init__(*options: dict[str, str]*, *columns: dict[str, ColumnDefinition]*) → None

Methods

__init__(options, columns)
execute(quals, columns[, sortkeys])
import_schema(schema, srv_options, options, ...)

gooddata_fdw.environment.Qual

gooddata_fdw.environment.**Qual**

alias of *QualStub*

gooddata_fdw.environment.QualStub

class gooddata_fdw.environment.**QualStub**(*field_name: str*, *operator: Union[str, tuple[str, str]]*, *value: Any*)

Bases: object

__init__(*field_name: str*, *operator: Union[str, tuple[str, str]]*, *value: Any*) → None

Methods

__init__(field_name, operator, value)

gooddata_fdw.environment.TableDefinition

gooddata_fdw.environment.**TableDefinition**

alias of *TableDefinitionStub*

gooddata_fdw.environment.TableDefinitionStub

class gooddata_fdw.environment.**TableDefinitionStub**(*table_name: str*, *columns: list[ColumnDefinition]*, *options: dict[str, str]*)

Bases: object

__init__(*table_name: str*, *columns: list[ColumnDefinition]*, *options: dict[str, str]*) → None

Methods

__init__(table_name, columns, options)

gooddata_fdw.executor

Classes

ComputeExecutor(inputs)
CustomExecutor(inputs)
Executor(inputs, column_validators)
ExecutorFactory()
InitData(sdk, server_options, table_options, ...)
InsightExecutor(inputs)

gooddata_fdw.executor.ComputeExecutor

class gooddata_fdw.executor.**ComputeExecutor**(*inputs:* InitData)

Bases: *Executor*

__init__(*inputs:* InitData) → None

Methods

__init__(inputs)
can_react(inputs)
execute(quals, columns[, sort_keys])
validate_columns_def()

gooddata_fdw.executor.CustomExecutor

class gooddata_fdw.executor.**CustomExecutor**(*inputs:* InitData)

Bases: *Executor*

__init__(*inputs:* InitData) → None

Methods

__init__(inputs)
can_react(inputs)
execute(quals, columns[, sort_keys])
validate_columns_def()

gooddata_fdw.executor.Executor

class gooddata_fdw.executor.**Executor**(*inputs:* InitData, *column_validators: list[col_val.ColumnValidator]*)

Bases: object

__init__(*inputs:* InitData, *column_validators: list[col_val.ColumnValidator]*) → None

Methods

__init__(inputs, column_validators)
can_react(inputs)
execute(quals, columns[, sort_keys])
validate_columns_def()

gooddata_fdw.executor.ExecutorFactory

class gooddata_fdw.executor.**ExecutorFactory**

Bases: object

__init__()

Methods

__init__()
create(inputs)

gooddata_fdw.executor.InitData

class gooddata_fdw.executor.**InitData**(*sdk*, *server_options*, *table_options*, *columns*)

Bases: tuple

__init__()

Methods

__init__()

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

Attributes

columns	Alias for field number 3
sdk	Alias for field number 0
server_options	Alias for field number 1
table_options	Alias for field number 2

property columns

Alias for field number 3

count(*value*, */*)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

property sdk

Alias for field number 0

property server_options
Alias for field number 1

property table_options
Alias for field number 2

gooddata_fdw.executor.InsightExecutor

class gooddata_fdw.executor.**InsightExecutor**(*inputs:* InitData)
Bases: *Executor*

__init__(*inputs:* InitData) → None

Methods

__init__(inputs)
can_react(inputs)
execute(quals, columns[, sort_keys])
validate_columns_def()

gooddata_fdw.fdw

Module Attributes

USER_AGENT	Extra segment of the User-Agent header that will be appended to standard gooddata-sdk user agent.

gooddata_fdw.fdw.USER_AGENT

gooddata_fdw.fdw.**USER_AGENT = 'gooddata-fdw/1.3.0'**
Extra segment of the User-Agent header that will be appended to standard gooddata-sdk user agent.

Classes

GoodDataForeignDataWrapper(options, columns)

gooddata_fdw.fdw.GoodDataForeignDataWrapper

class gooddata_fdw.fdw.**GoodDataForeignDataWrapper**(*options: dict[str, str]*, *columns: dict[str, ColumnDefinition]*)

Bases: *ForeignDataWrapperStub*

__init__(*options: dict[str, str]*, *columns: dict[str, ColumnDefinition]*) → None

Methods

__init__(options, columns)	
delete(oldvalues)	
execute(quals, columns[, sortkeys])	
import_schema(schema, srv_options, options, ...)	
insert(values)	
update(oldvalues, newvalues)	

Attributes

rowid_column

gooddata_fdw.filter

Functions

extract_filters_from_quals(quals, table_columns)	Convert quals to filters.

gooddata_fdw.filter.extract_filters_from_quals

gooddata_fdw.filter.**extract_filters_from_quals**(*quals: list[Qual]*, *table_columns: dict[str, ColumnDefinition]*) → list[Filter]

Convert quals to filters. Now only simple attribute filters are supported.

Parameters

- **quals** – multicorn quals representing filters in SQL WHERE clause
- **table_columns** – list of table columns

Returns

list of filters

gooddata_fdw.import_workspace

Classes

ImporterInitData(sdk, workspace, ...)	
InsightsWorkspaceImporter(data)	
SemanticLayerWorkspaceImporter(data)	
WorkspaceImporter(data)	
WorkspaceImportersLocator()	

gooddata_fdw.import_workspace.ImporterInitData

class gooddata_fdw.import_workspace.**ImporterInitData**(*sdk*, *workspace*, *server_options*, *import_options*, *restriction_type*, *restricts*)

Bases: `tuple`

__init__()

Methods

__init__()	
count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

Attributes

import_options	Alias for field number 3
restriction_type	Alias for field number 4
restricts	Alias for field number 5
sdk	Alias for field number 0
server_options	Alias for field number 2
workspace	Alias for field number 1

count(*value*, */*)

Return number of occurrences of value.

property import_options

Alias for field number 3

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

property restriction_type
Alias for field number 4

property restricts
Alias for field number 5

property sdk
Alias for field number 0

property server_options
Alias for field number 2

property workspace
Alias for field number 1

gooddata_fdw.import_workspace.InsightsWorkspaceImporter

class gooddata_fdw.import_workspace.**InsightsWorkspaceImporter**(*data:* ImporterInitData)
Bases: *WorkspaceImporter*

__init__(*data:* ImporterInitData) → None

Methods

__init__(data)
import_tables()
support_object_type(object_type)

gooddata_fdw.import_workspace.SemanticLayerWorkspaceImporter

class gooddata_fdw.import_workspace.**SemanticLayerWorkspaceImporter**(*data:* ImporterInitData)
Bases: *WorkspaceImporter*

__init__(*data:* ImporterInitData) → None

Methods

__init__(data)
import_tables()
support_object_type(object_type)

gooddata_fdw.import_workspace.WorkspaceImporter

class gooddata_fdw.import_workspace.**WorkspaceImporter**(*data:* ImporterInitData)

Bases: object

__init__(*data:* ImporterInitData) → None

Methods

__init__(data)
import_tables()
support_object_type(object_type)

gooddata_fdw.import_workspace.WorkspaceImportersLocator

class gooddata_fdw.import_workspace.**WorkspaceImportersLocator**

Bases: object

__init__()

Methods

__init__()
locate(object_type)
register(class_)

gooddata_fdw.naming

Classes

CatalogNamingStrategy()
DefaultCatalogNamingStrategy()
DefaultInsightColumnNaming()
DefaultInsightTableNaming()
InsightColumnNamingStrategy()
InsightTableNamingStrategy()

gooddata_fdw.naming.CatalogNamingStrategy

class gooddata_fdw.naming.**CatalogNamingStrategy**

Bases: object

__init__()

Methods

__init__()
col_name_for_fact(attr)
col_name_for_label(attr)
col_name_for_metric(attr)

gooddata_fdw.naming.DefaultCatalogNamingStrategy

class gooddata_fdw.naming.**DefaultCatalogNamingStrategy**

Bases: object

__init__() → None

Methods

__init__()
col_name_for_fact(fact, dataset)
col_name_for_label(label, dataset)
col_name_for_metric(metric)

gooddata_fdw.naming.DefaultInsightColumnNaming

class gooddata_fdw.naming.**DefaultInsightColumnNaming**

Bases: *InsightColumnNamingStrategy*

__init__() → None

Methods

__init__()
col_name_for_attribute(attr)
col_name_for_metric(metric)

gooddata_fdw.naming.DefaultInsightTableNaming

class gooddata_fdw.naming.**DefaultInsightTableNaming**

Bases: *InsightTableNamingStrategy*

__init__() → None

Methods

__init__()
table_name_for_insight(insight)

gooddata_fdw.naming.InsightColumnNamingStrategy

class gooddata_fdw.naming.**InsightColumnNamingStrategy**

Bases: object

__init__()

Methods

__init__()
col_name_for_attribute(attr)
col_name_for_metric(attr)

gooddata_fdw.naming.InsightTableNamingStrategy

class gooddata_fdw.naming.**InsightTableNamingStrategy**

Bases: object

__init__()

Methods

__init__()
table_name_for_insight(insight)

gooddata_fdw.options

Classes

BaseOptions([validate, skip_attributes])
ImportSchemaOptions(options)
ServerOptions(options)
TableOptions(options)

gooddata_fdw.options.BaseOptions

class gooddata_fdw.options.**BaseOptions**(*validate: bool = True*, *skip_attributes: Optional[list[str]] = None*)

Bases: object

__init__(*validate: bool = True*, *skip_attributes: Optional[list[str]] = None*) → None

Methods

__init__([validate, skip_attributes])

gooddata_fdw.options.ImportSchemaOptions

class gooddata_fdw.options.**ImportSchemaOptions**(*options: dict[str, str]*)

Bases: *BaseOptions*

__init__(*options: dict[str, str]*) → None

Methods

__init__(options)
metric_data_type([precision])

Attributes

METRIC_DIGITS_AFTER_DEC_POINT_DEFAULT
METRIC_DIGITS_BEFORE_DEC_POINT_DEFAULT
numeric_max_size
object_type

gooddata_fdw.options.ServerOptions

class gooddata_fdw.options.**ServerOptions**(*options: dict[str, str]*)

Bases: *BaseOptions*

__init__(*options: dict[str, str]*) → None

Methods

__init__(options)

Attributes

headers_host
host
token

gooddata_fdw.options.TableOptions

class gooddata_fdw.options.**TableOptions**(*options: dict[str, str]*)

Bases: *BaseOptions*

__init__(*options: dict[str, str]*) → None

Methods

__init__(options)

Attributes

compute
insight
workspace

gooddata_fdw.pg_logging

gooddata_fdw.result_reader

Classes

InsightTableResultReader(table_columns, ...)
TableResultReader(table_columns)

gooddata_fdw.result_reader.InsightTableResultReader

class gooddata_fdw.result_reader.**InsightTableResultReader**(*table_columns: dict[str, ColumnDefinition]*, *query_columns: list[str]*)

Bases: *TableResultReader*

__init__(*table_columns: dict[str, ColumnDefinition]*, *query_columns: list[str]*) → None

Methods

__init__(table_columns, query_columns)
read_all_rows(table)

gooddata_fdw.result_reader.TableResultReader

class gooddata_fdw.result_reader.**TableResultReader**(*table_columns: dict[str, ColumnDefinition]*)

Bases: object

__init__(*table_columns: dict[str, ColumnDefinition]*) → None

Methods

__init__(table_columns)
read_all_rows(table)

PYTHON MODULE INDEX

g

INDEX

Symbols

B

C

D

E

F

G

I

L

M

Q

R

S

T

U

V

W