GoodData Foreign Data Wrapper

Release 0.7.0

GoodData Corporation

Jul 14, 2022

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

GETTING STARTED

1.1 Requirements

- GoodData.CN installation; either running on your cloud infrastructure or the free Community Edition running on your workstation
- Python 3.7 or newer

1.2 Installation

For convenience a `Dockerfile` is already in place which, when started, will run `PostgreSQL 12` with `multicorn` and `gooddata-fdw` pre-installed.

For an even better user experience we prepared a `docker-compose.yaml` file which contains both the `gooddata-fdw` and `gooddata-cn-ce` services.

If you execute (in repository root folder):

```
docker-compose up -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`.

You can also execute:

```
docker-compose build
```

to rebuild the Foreign Data Wrapper image.

If you would like to purge a container completely (including the volume) and start from scratch, you can use a helper script:

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

1.3 Add Your 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 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.

CHAPTER

TWO

USAGE

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
    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).

2.1 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
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.

2.2 Custom Reports as 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.

2.3 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`.

2.4 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.

CHAPTER

THREE

API DOCUMENTATION

3.1 API

gooddata_fdw	
gooddata_sdk	The *gooddata-sdk* package aims to provide clean and convenient Python APIs to interact with GoodData.CN.

3.1.1 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[*gooddata_sdk.catalog.CatalogAttribute*]*) → str

Determine what postgres type should be used for *attribute*. :param attribute: catalog attribute instance

gooddata_fdw.column_utils.table_col_as_computable

gooddata_fdw.column_utils.**table_col_as_computable**(*col:* gooddata_fdw.environment.ColumnDefinitionStub) → Union[*gooddata_sdk.compute_model.Attribute*, *gooddata_sdk.compute_model.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: *gooddata_fdw.column_validation.ColumnValidator*

__init__(*mandatory: bool*)

Methods

__init__(mandatory)
validate(column_name, column_def)

gooddata_fdw.column_validation.LocalIdOptionValidator

class gooddata_fdw.column_validation.**LocalIdOptionValidator**
Bases: *gooddata_fdw.column_validation.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 *gooddata_fdw.environment.ColumnDefinitionStub*
ColumnDefinitionStub(column_name, type_name, ...)	
ForeignDataWrapper	alias of *gooddata_fdw.environment.ForeignDataWrapperStub*
ForeignDataWrapperStub(options, columns)	
Qual	alias of *gooddata_fdw.environment.QualStub*
QualStub(field_name, operator, value)	
TableDefinition	alias of *gooddata_fdw.environment.TableDefinitionStub*
TableDefinitionStub(table_name, columns, options)	

gooddata_fdw.environment.ColumnDefinition

gooddata_fdw.environment.**ColumnDefinition**
alias of *gooddata_fdw.environment.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 *gooddata_fdw.environment.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 *gooddata_fdw.environment.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 *gooddata_fdw.environment.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)	

continues on next page

Table 15 – continued from previous page

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

gooddata_fdw.executor.ComputeExecutor

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

Bases: *gooddata_fdw.executor.Executor*

__init__(*inputs:* gooddata_fdw.executor.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:* gooddata_fdw.executor.InitData)

Bases: *gooddata_fdw.executor.Executor*

__init__(*inputs:* gooddata_fdw.executor.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:* gooddata_fdw.executor.InitData)
Bases: *gooddata_fdw.executor.Executor*

__init__(*inputs:* gooddata_fdw.executor.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/0.7.0'**

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: *gooddata_fdw.environment.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:* gooddata_fdw.import_workspace.ImporterInitData)

Bases: *gooddata_fdw.import_workspace.WorkspaceImporter*

__init__(*data:* gooddata_fdw.import_workspace.ImporterInitData) → None

Methods

__init__(data)
import_tables()
support_object_type(object_type)

gooddata_fdw.import_workspace.SemanticLayerWorkspaceImporter

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

Bases: *gooddata_fdw.import_workspace.WorkspaceImporter*

__init__(*data:* gooddata_fdw.import_workspace.ImporterInitData) → None

Methods

__init__(data)
import_tables()
support_object_type(object_type)

gooddata_fdw.import_workspace.WorkspaceImporter

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

Bases: object

__init__(*data:* gooddata_fdw.import_workspace.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: *gooddata_fdw.naming.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: *gooddata_fdw.naming.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: *gooddata_fdw.options.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: *gooddata_fdw.options.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: *gooddata_fdw.options.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: *gooddata_fdw.result_reader.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)	

3.1.2 gooddata_sdk

The *gooddata-sdk* package aims to provide clean and convenient Python APIs to interact with GoodData.CN.

At the moment the SDK provides services to inspect and interact with the Semantic Model and consume analytics.

Modules

gooddata_sdk.catalog	
gooddata_sdk.client	Module containing a class that provides access to metadata and afm services.
gooddata_sdk.compute	
gooddata_sdk.compute_model	
gooddata_sdk.insight	
gooddata_sdk.sdk	
gooddata_sdk.table	
gooddata_sdk.type_converter	
gooddata_sdk.utils	

gooddata_sdk.catalog

Classes

Catalog(valid_obj_fun, datasets, metrics)
CatalogAttribute(attribute, labels)
CatalogDataset(dataset, attributes, facts)
CatalogEntry()
CatalogFact(fact)
CatalogLabel(label)
CatalogMetric(metric)
CatalogService(api_client)

gooddata_sdk.catalog.Catalog

class gooddata_sdk.catalog.**Catalog**(*valid_obj_fun: functools.partial[dict[str, set[str]]]*, *datasets: list[*CatalogDataset*]*, *metrics: list[*CatalogMetric*]*)

Bases: `object`

__init__(*valid_obj_fun: functools.partial[dict[str, set[str]]]*, *datasets: list[*CatalogDataset*]*, *metrics: list[*CatalogMetric*]*) → None

Methods

__init__(valid_obj_fun, datasets, metrics)

catalog_with_valid_objects(ctx)	Returns a new instance of catalog which contains only those datasets (attributes and facts) that are valid in the provided context.
find_label_attribute(id_obj)	Get attribute by label id.
get_dataset(dataset_id)	Gets dataset by id.
get_metric(metric_id)	Gets metric by id.

Attributes

datasets
metrics

catalog_with_valid_objects(*ctx: Union[*gooddata_sdk.compute_model.Attribute, gooddata_sdk.compute_model.Metric, gooddata_sdk.compute_model.Filter, gooddata_sdk.catalog.CatalogLabel, gooddata_sdk.catalog.CatalogFact, gooddata_sdk.catalog.CatalogMetric, *List[Union[*gooddata_sdk.compute_model.Attribute, gooddata_sdk.compute_model.Metric, gooddata_sdk.compute_model.Filter, gooddata_sdk.catalog.CatalogLabel, gooddata_sdk.catalog.CatalogFact, gooddata_sdk.catalog.CatalogMetric*]],* gooddata_sdk.compute.ExecutionDefinition*]*) → *gooddata_sdk.catalog.Catalog*

Returns a new instance of catalog which contains only those datasets (attributes and facts) that are valid in the provided context. The context is composed of one more more entities of the semantic model and the filtered catalog will contain only those entities that can be safely added on top of that existing context.

Parameters **ctx** – existing context. you can specify context in one of the following ways: - single item or list of items from the execution model - single item or list of items from catalog model; catalog fact, label or metric may be added - the entire execution definition that is used to compute analytics

Returns

find_label_attribute(*id_obj: Union[str,* gooddata_sdk.compute_model.ObjId*, Dict[str, Dict[str, str]], Dict[str, str]]*) → Optional[*gooddata_sdk.catalog.CatalogAttribute*]

Get attribute by label id.

get_dataset(*dataset_id: Union[str,* gooddata_sdk.compute_model.ObjId*]*) → Optional[*gooddata_sdk.catalog.CatalogDataset*]

Gets dataset by id. The id can be either an instance of ObjId or string containing serialized ObjId ('dataset/some.dataset.id') or contain just the id part ('some.dataset.id').

Parameters **dataset_id** – fully qualified dataset entity id (type/id) or just the identifier of dataset entity

Returns instance of CatalogDataset or None if no such dataset in catalog

:rtype CatalogDataset

get_metric(*metric_id: Union[str,* gooddata_sdk.compute_model.ObjId*]*) → Optional[*gooddata_sdk.catalog.CatalogMetric*]

Gets metric by id. The id can be either an instance of ObjId or string containing serialized ObjId ('metric/some.metric.id') or contain just the id part ('some.metric.id').

Parameters **metric_id** – fully qualified metric entity id (type/id) or just the identifier of metric entity

Returns instance of CatalogMetric or None if no such metric in catalog

:rtype CatalogMetric

gooddata_sdk.catalog.CatalogAttribute

class gooddata_sdk.catalog.**CatalogAttribute**(*attribute: dict[str, Any]*, *labels: list[*CatalogLabel*]*)

Bases: *gooddata_sdk.catalog.CatalogEntry*

__init__(*attribute: dict[str, Any]*, *labels: list[*CatalogLabel*]*) → None

Methods

__init__(attribute, labels)
as_computable()
find_label(id_obj)
primary_label()

Attributes

dataset
description
granularity
id
labels
obj_id
title
type

gooddata_sdk.catalog.CatalogDataset

class gooddata_sdk.catalog.**CatalogDataset**(*dataset: dict[str, Any]*, *attributes: list[*CatalogAttribute*]*, *facts: list[*CatalogFact*]*)

Bases: *gooddata_sdk.catalog.CatalogEntry*

__init__(*dataset: dict[str, Any]*, *attributes: list[*CatalogAttribute*]*, *facts: list[*CatalogFact*]*) → None

Methods

__init__(dataset, attributes, facts)	
filter_dataset(valid_objects)	Filters dataset so that it contains only attributes and facts that are part of the provided valid objects structure.
find_label_attribute(id_obj)	

Attributes

attributes
data_type
description
facts
id
obj_id
title
type

filter_dataset(*valid_objects: Dict[str, Set[str]]*) → Optional[*gooddata_sdk.catalog.CatalogDataset*]

Filters dataset so that it contains only attributes and facts that are part of the provided valid objects structure.

Parameters **valid_objects** – mapping of object type to a set of valid object ids

Returns CatalogDataset containing only valid attributes and facts; None if all of the attributes and facts were filtered out

gooddata_sdk.catalog.CatalogEntry

class gooddata_sdk.catalog.**CatalogEntry**
Bases: object

__init__()

Methods

__init__()

Attributes

description
id
obj_id
title
type

gooddata_sdk.catalog.CatalogFact

class gooddata_sdk.catalog.**CatalogFact**(*fact: dict[str, Any]*)
Bases: *gooddata_sdk.catalog.CatalogEntry*

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

Methods

__init__(fact)
as_computable()

Attributes

description
id
obj_id
title
type

gooddata_sdk.catalog.CatalogLabel

class gooddata_sdk.catalog.**CatalogLabel**(*label: dict[str, Any]*)

Bases: *gooddata_sdk.catalog.CatalogEntry*

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

Methods

__init__(label)
as_computable()

Attributes

description
id
obj_id
primary
title
type

gooddata_sdk.catalog.CatalogMetric

class gooddata_sdk.catalog.**CatalogMetric**(*metric: dict[str, Any]*)

Bases: *gooddata_sdk.catalog.CatalogEntry*

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

Methods

__init__(metric)
as_computable()

Attributes

description
format
id
obj_id
title
type

gooddata_sdk.catalog.CatalogService

class gooddata_sdk.catalog.**CatalogService**(*api_client:* gooddata_sdk.client.GoodDataApiClient)

Bases: object

__init__(*api_client:* gooddata_sdk.client.GoodDataApiClient) → None

Methods

__init__(api_client)	
compute_valid_objects(workspace_id, ctx)	Returns attributes, facts, and metrics which are valid to add to a context that already contains some entities from the semantic model.
get_full_catalog(workspace_id)	Retrieves catalog for a workspace.

compute_valid_objects(*workspace_id: str*, *ctx: Union[*gooddata_sdk.compute_model.Attribute, gooddata_sdk.compute_model.Metric, gooddata_sdk.compute_model.Filter, gooddata_sdk.catalog.CatalogLabel, gooddata_sdk.catalog.CatalogFact, gooddata_sdk.catalog.CatalogMetric, *List[Union[*gooddata_sdk.compute_model.Attribute, gooddata_sdk.compute_model.Metric, gooddata_sdk.compute_model.Filter, gooddata_sdk.catalog.CatalogLabel, gooddata_sdk.catalog.CatalogFact, gooddata_sdk.catalog.CatalogMetric*]],* gooddata_sdk.compute.ExecutionDefinition*]*) → Dict[str, Set[str]]

Returns attributes, facts, and metrics which are valid to add to a context that already contains some entities from the semantic model. The entities are typically used to compute analytics and come from the execution definition. You may, however, specify the entities through different layers of convenience.

Parameters

- **workspace_id** – workspace identifier
- **ctx** – items already in context. you can specify context in one of the following ways: - single item or list of items from the execution model - single item or list of items from catalog model; catalog fact, label or metric may be added - the entire execution definition that is used to compute analytics

Returns a dict of sets; type of available object is used as key in the dict, the value is a set containing id's of available items

get_full_catalog(*workspace_id: str*) → *gooddata_sdk.catalog.Catalog*

Retrieves catalog for a workspace. Catalog contains all data sets and metrics defined in that workspace.

Parameters **workspace_id** – workspace identifier

Returns

gooddata_sdk.client

Module containing a class that provides access to metadata and afm services.

Classes

GoodDataApiClient(host, token[, ...])	Provide access to metadata and afm services.

gooddata_sdk.client.GoodDataApiClient

class gooddata_sdk.client.**GoodDataApiClient**(*host: str*, *token: str*, *custom_headers: Optional[dict[str, str]] = None*, *extra_user_agent: Optional[str] = None*)

Bases: object

Provide access to metadata and afm services.

__init__(*host: str*, *token: str*, *custom_headers: Optional[dict[str, str]] = None*, *extra_user_agent: Optional[str] = None*) → None

Take url, token for connecting to GoodData.CN.

HTTP requests made by this class may be enriched by *custom_headers* dict containing header names as keys and header values as dict values.

extra_user_agent is optional string to be added to default http User-Agent header. This takes precedence over custom_headers setting.

Methods

__init__(host, token[, custom_headers, ...])	Take url, token for connecting to GoodData.CN.

Attributes

afm_client
metadata_client
scan_client

gooddata_sdk.compute

Classes

ComputeService(api_client)	Compute service drives computation of analytics for a GoodData.CN workspaces.
ExecutionDefinition(attributes, metrics, ...)	
ExecutionResponse(actions_api, workspace_id, ...)	
ExecutionResult(result)	

gooddata_sdk.compute.ComputeService

class gooddata_sdk.compute.**ComputeService**(*api_client:* gooddata_sdk.client.GoodDataApiClient)

Bases: object

Compute service drives computation of analytics for a GoodData.CN workspaces. The prescription of what to compute is encapsulated by the ExecutionDefinition which consists of attributes, metrics, filters and definition of dimensions that influence how to organize the data in the result.

__init__(*api_client:* gooddata_sdk.client.GoodDataApiClient)

Methods

__init__(api_client)	
for_exec_def(workspace_id, exec_def)	Starts computation in GoodData.CN workspace, using the provided execution definition.

for_exec_def(*workspace_id: str*, *exec_def:* gooddata_sdk.compute.ExecutionDefinition) → *gooddata_sdk.compute.ExecutionResponse*

Starts computation in GoodData.CN workspace, using the provided execution definition.

Parameters

- **workspace_id** – workspace identifier
- **exec_def** – execution definition - this prescribes what to calculate, how to place labels and metric values into dimensions

Returns

gooddata_sdk.compute.ExecutionDefinition

class gooddata_sdk.compute.**ExecutionDefinition**(*attributes: Optional[list[Attribute]]*, *metrics: Optional[list[Metric]]*, *filters: Optional[list[Filter]]*, *dimensions: list[Optional[list[str]]]*)

Bases: object

__init__(*attributes: Optional[list[Attribute]]*, *metrics: Optional[list[Metric]]*, *filters: Optional[list[Filter]]*, *dimensions: list[Optional[list[str]]]*) → None

Methods

__init__(attributes, metrics, filters, ...)	
as_api_model()	
has_attributes()	
has_filters()	
has_metrics()	
is_one_dim()	
is_two_dim()	

Attributes

attributes
dimensions
filters
metrics

gooddata_sdk.compute.ExecutionResponse

class gooddata_sdk.compute.**ExecutionResponse**(*actions_api: gooddata_afm_client.api.actions_api.ActionsApi*, *workspace_id: str*, *exec_def:* gooddata_sdk.compute.ExecutionDefinition, *response: gooddata_afm_client.model.afm_execution_response.AfmExecutionResponse*)

Bases: object

__init__(*actions_api: gooddata_afm_client.api.actions_api.ActionsApi*, *workspace_id: str*, *exec_def:* gooddata_sdk.compute.ExecutionDefinition, *response: gooddata_afm_client.model.afm_execution_response.AfmExecutionResponse*)

Methods

__init__(actions_api, workspace_id, ...)	
read_result(limit[, offset])	Reads from the execution result.

Attributes

exec_def
result_id
workspace_id

read_result(*limit: Union[int, list[int]]*, *offset: Union[None, int, list[int]] = None*) → *ExecutionResult*

Reads from the execution result. :param offset: :param limit: :return:

gooddata_sdk.compute.ExecutionResult

class gooddata_sdk.compute.**ExecutionResult**(*result: gooddata_afm_client.model.execution_result.ExecutionResult*)

Bases: object

__init__(*result: gooddata_afm_client.model.execution_result.ExecutionResult*)

Methods

__init__(result)	
get_all_header_values(dim, header_idx)	
is_complete([dim])	
next_page_start([dim])	

Attributes

data	
grand_totals	
headers	
paging	
paging_count	
paging_offset	
paging_total	

gooddata_sdk.compute_model

Functions

compute_model_to_api_model([attributes, ...])	Transforms categorized execution model entities (attributes, metrics, facts) into an API model that can be used for computations of data results or computations of object availability.

gooddata_sdk.compute_model.compute_model_to_api_model

gooddata_sdk.compute_model.**compute_model_to_api_model**(*attributes: Optional[list[Attribute]] = None*, *metrics: Optional[list[Metric]] = None*, *filters: Optional[list[Filter]] = None*) → afm_models.AFM

Transforms categorized execution model entities (attributes, metrics, facts) into an API model that can be used for computations of data results or computations of object availability.

Parameters

- **attributes** – optionally specify list of attributes
- **metrics** – optionally specify list of metrics
- **filters** – optionally specify list of filters

Returns

Classes

AbsoluteDateFilter(dataset, from_date, to_date)	
AllTimeFilter()	Filter that is semantically equivalent to absent filter.
ArithmeticMetric(local_id, operator, operands)	
Attribute(local_id, label)	
AttributeFilter(label[, values])	
ExecModelEntity()	
Filter()	
Metric(local_id)	
MetricValueFilter(metric, operator, values)	
NegativeAttributeFilter(label[, values])	
ObjId(id, type)	
PopDate(attribute, periods_ago)	
PopDateDataset(dataset, periods_ago)	
PopDateMetric(local_id, metric, date_attributes)	
PopDatesetMetric(local_id, metric, date_datasets)	
PositiveAttributeFilter(label[, values])	

continues on next page

Table 82 – continued from previous page

RankingFilter(metrics, operator, value, ...)	
RelativeDateFilter(dataset, granularity, ...)	
SimpleMetric(local_id, item[, aggregation, ...])	

gooddata_sdk.compute_model.AbsoluteDateFilter

class gooddata_sdk.compute_model.**AbsoluteDateFilter**(*dataset:* gooddata_sdk.compute_model.ObjId, *from_date: str*, *to_date: str*)

Bases: *gooddata_sdk.compute_model.Filter*

__init__(*dataset:* gooddata_sdk.compute_model.ObjId, *from_date: str*, *to_date: str*) → None

Methods

__init__(dataset, from_date, to_date)	
as_api_model()	
is_noop()	

Attributes

apply_on_result
dataset
from_date
to_date

gooddata_sdk.compute_model.AllTimeFilter

class gooddata_sdk.compute_model.**AllTimeFilter**

Bases: *gooddata_sdk.compute_model.Filter*

Filter that is semantically equivalent to absent filter.

This filter exists because 'All time filter' retrieved from GoodData.CN is non-standard as it does not have *from* and *to* fields; this is also the reason why as_api_model method is not implemented - it would lead to invalid object.

The main feature of this filter is noop.

__init__() → None

Methods

__init__()
as_api_model()
is_noop()

Attributes

apply_on_result

gooddata_sdk.compute_model.ArithmeticMetric

class gooddata_sdk.compute_model.**ArithmeticMetric**(*local_id: str*, *operator: str*, *operands: list[Union[str,* Metric*]]*)

Bases: *gooddata_sdk.compute_model.Metric*

__init__(*local_id: str*, *operator: str*, *operands: list[Union[str,* Metric*]]*) → None

Methods

__init__(local_id, operator, operands)
as_api_model()

Attributes

local_id
operand_local_ids
operator

gooddata_sdk.compute_model.Attribute

class gooddata_sdk.compute_model.**Attribute**(*local_id: str*, *label: Union[gooddata_sdk.compute_model.ObjId, str]*)

Bases: *gooddata_sdk.compute_model.ExecModelEntity*

__init__(*local_id: str*, *label: Union[gooddata_sdk.compute_model.ObjId, str]*) → None

Creates new attribute that can be used to slice or dice metric values during computation.

Parameters

- **local_id** – identifier of the attribute within the execution
- **label** – identifier of the label to use for slicing or dicing; specified either as ObjId or str containing the label id

Methods

__init__(local_id, label)	Creates new attribute that can be used to slice or dice metric values during computation.
as_api_model()	
has_same_label(other)	

Attributes

label
local_id

gooddata_sdk.compute_model.AttributeFilter

class gooddata_sdk.compute_model.**AttributeFilter**(*label: Union[ObjId, str, Attribute]*, *values: list[str] = None*)

Bases: *gooddata_sdk.compute_model.Filter*

__init__(*label: Union[ObjId, str, Attribute]*, *values: list[str] = None*) → None

Methods

__init__(label[, values])
as_api_model()
is_noop()

Attributes

apply_on_result
label
values

gooddata_sdk.compute_model.ExecModelEntity

class gooddata_sdk.compute_model.**ExecModelEntity**
Bases: object

__init__() → None

Methods

__init__()
as_api_model()

gooddata_sdk.compute_model.Filter

class gooddata_sdk.compute_model.**Filter**
Bases: *gooddata_sdk.compute_model.ExecModelEntity*

__init__() → None

Methods

__init__()
as_api_model()
is_noop()

Attributes

apply_on_result

gooddata_sdk.compute_model.Metric

class gooddata_sdk.compute_model.**Metric**(*local_id: str*)

Bases: *gooddata_sdk.compute_model.ExecModelEntity*

__init__(*local_id: str*) → None

Methods

__init__(local_id)
as_api_model()

Attributes

local_id

gooddata_sdk.compute_model.MetricValueFilter

class gooddata_sdk.compute_model.**MetricValueFilter**(*metric: Union[ObjId, str, Metric]*, *operator: str*, *values: Union[float, int, tuple[float, float]]*, *treat_nulls_as: Union[float, None] = None*)

Bases: *gooddata_sdk.compute_model.Filter*

__init__(*metric: Union[ObjId, str, Metric]*, *operator: str*, *values: Union[float, int, tuple[float, float]]*, *treat_nulls_as: Union[float, None] = None*) → None

Methods

__init__(metric, operator, values[, ...])
as_api_model()
is_noop()

Attributes

apply_on_result
metric
operator
treat_nulls_as
values

gooddata_sdk.compute_model.NegativeAttributeFilter

class gooddata_sdk.compute_model.**NegativeAttributeFilter**(*label: Union[ObjId, str, Attribute]*, *values: list[str] = None*)

Bases: *gooddata_sdk.compute_model.AttributeFilter*

__init__(*label: Union[ObjId, str, Attribute]*, *values: list[str] = None*) → None

Methods

__init__(label[, values])
as_api_model()
is_noop()

Attributes

apply_on_result
label
values

gooddata_sdk.compute_model.ObjId

class gooddata_sdk.compute_model.**ObjId**(*id: str*, *type: str*)

Bases: object

__init__(*id: str*, *type: str*) → None

Methods

__init__(id, type)
as_afm_id()
as_identifier()

Attributes

id
type

gooddata_sdk.compute_model.PopDate

class gooddata_sdk.compute_model.**PopDate**(*attribute: Union[*gooddata_sdk.compute_model.ObjId, gooddata_sdk.compute_model.Attribute*]*, *periods_ago: int*)

Bases: object

__init__(*attribute: Union[*gooddata_sdk.compute_model.ObjId, gooddata_sdk.compute_model.Attribute*]*, *periods_ago: int*) → None

Methods

__init__(attribute, periods_ago)
as_api_model()

Attributes

attribute
periods_ago

gooddata_sdk.compute_model.PopDateDataset

class gooddata_sdk.compute_model.**PopDateDataset**(*dataset: Union[*gooddata_sdk.compute_model.ObjId*, str]*, *periods_ago: int*)

Bases: object

__init__(*dataset: Union[*gooddata_sdk.compute_model.ObjId*, str]*, *periods_ago: int*) → None

Methods

__init__(dataset, periods_ago)
as_api_model()

Attributes

dataset
periods_ago

gooddata_sdk.compute_model.PopDateMetric

class gooddata_sdk.compute_model.**PopDateMetric**(*local_id: str*, *metric: Union[str,* Metric*]*, *date_attributes: list[*PopDate*]*)

Bases: *gooddata_sdk.compute_model.Metric*

__init__(*local_id: str*, *metric: Union[str,* Metric*]*, *date_attributes: list[*PopDate*]*) → None

Methods

__init__(local_id, metric, date_attributes)
as_api_model()

Attributes

date_attributes
local_id
metric_local_id

gooddata_sdk.compute_model.PopDatesetMetric

class gooddata_sdk.compute_model.**PopDatesetMetric**(*local_id: str*, *metric: Union[str,* Metric*]*, *date_datasets: list[*PopDateDataset*]*)

Bases: *gooddata_sdk.compute_model.Metric*

__init__(*local_id: str*, *metric: Union[str,* Metric*]*, *date_datasets: list[*PopDateDataset*]*) → None

Methods

__init__(local_id, metric, date_datasets)
as_api_model()

Attributes

date_datasets
local_id
metric_local_id

gooddata_sdk.compute_model.PositiveAttributeFilter

class gooddata_sdk.compute_model.**PositiveAttributeFilter**(*label: Union[*ObjId*, str,* Attribute*]*, *values: list[str] = None*)

Bases: *gooddata_sdk.compute_model.AttributeFilter*

__init__(*label: Union[*ObjId*, str,* Attribute*]*, *values: list[str] = None*) → None

Methods

__init__(label[, values])
as_api_model()
is_noop()

Attributes

apply_on_result
label
values

gooddata_sdk.compute_model.RankingFilter

class gooddata_sdk.compute_model.**RankingFilter**(*metrics: list[Union[*ObjId, Metric, *str]]*, *operator: str*, *value: int*, *dimensionality: Optional[list[Union[str,* ObjId, Attribute, Metric*]]]*)

Bases: *gooddata_sdk.compute_model.Filter*

__init__(*metrics: list[Union[*ObjId, Metric, *str]]*, *operator: str*, *value: int*, *dimensionality: Optional[list[Union[str,* ObjId, Attribute, Metric*]]]*) → None

Methods

__init__(metrics, operator, value, ...)
as_api_model()
is_noop()

Attributes

apply_on_result
dimensionality
metrics

continues on next page

Table 115 – continued from previous page

operator
value

gooddata_sdk.compute_model.RelativeDateFilter

class gooddata_sdk.compute_model.**RelativeDateFilter**(*dataset:* gooddata_sdk.compute_model.ObjId, *granularity: str*, *from_shift: int*, *to_shift: int*)

Bases: *gooddata_sdk.compute_model.Filter*

__init__(*dataset:* gooddata_sdk.compute_model.ObjId, *granularity: str*, *from_shift: int*, *to_shift: int*) → None

Methods

__init__(dataset, granularity, from_shift, ...)
as_api_model()
is_noop()

Attributes

apply_on_result
dataset
from_shift
granularity
to_shift

gooddata_sdk.compute_model.SimpleMetric

class gooddata_sdk.compute_model.**SimpleMetric**(*local_id: str*, *item:* ObjId, *aggregation: Optional[str] = None*, *compute_ratio: bool = False*, *filters: list[*Filter*] = None*)

Bases: *gooddata_sdk.compute_model.Metric*

__init__(*local_id: str*, *item:* ObjId, *aggregation: Optional[str] = None*, *compute_ratio: bool = False*, *filters: list[*Filter*] = None*) → None

Methods

__init__(local_id, item[, aggregation, ...])
as_api_model()

Attributes

aggregation
compute_ratio
filters
item
local_id

gooddata_sdk.insight

Classes

Insight(from_vis_obj[, side_loads])	
InsightAttribute(attribute)	
InsightBucket(bucket)	
InsightFilter(f)	
InsightMetric(metric)	Represents metric placed on an insight.
InsightService(api_client)	Insight Service allows retrieval of insights from a GD.CN workspace.

gooddata_sdk.insight.Insight

class gooddata_sdk.insight.**Insight**(*from_vis_obj: dict[str, Any]*, *side_loads: Optional[SideLoads] = None*)

Bases: object

__init__(*from_vis_obj: dict[str, Any]*, *side_loads: Optional[SideLoads] = None*) → None

Methods

`__init__`(from_vis_obj[, side_loads])
`get_metadata`(id_obj)

Attributes

`are_relations_valid`
`attributes`
`buckets`
`description`
`filters`
`id`
`metrics`
`properties`
`side_loads`
`sorts`
`title`
`vis_url`

gooddata_sdk.insight.InsightAttribute

class gooddata_sdk.insight.**InsightAttribute**(*attribute: dict[str, Any]*)

Bases: `object`

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

Methods

__init__(attribute)
as_computable()

Attributes

alias
label
label_id
local_id

gooddata_sdk.insight.InsightBucket

class gooddata_sdk.insight.**InsightBucket**(*bucket: dict[str, Any]*)

Bases: object

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

Methods

__init__(bucket)

Attributes

attributes
items
local_id
metrics

gooddata_sdk.insight.InsightFilter

class gooddata_sdk.insight.**InsightFilter**(*f: dict[str, Any]*)

Bases: `object`

__init__(*f: dict[str, Any]*) → None

Methods

__init__(f)	
as_computable()	

gooddata_sdk.insight.InsightMetric

class gooddata_sdk.insight.**InsightMetric**(*metric: dict[str, Any]*)

Bases: `object`

Represents metric placed on an insight.

Note: this has different shape than object passed to execution.

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

Methods

__init__(metric)	
as_computable()	

Attributes

alias	
format	
is_time_comparison	
item	
item_id	
local_id	
time_comparison_master	If this is a time comparison metric, return local_id of the master metric from which it is derived.

continues on next page

Table 129 – continued from previous page

title

property time_comparison_master: Optional[str]
If this is a time comparison metric, return local_id of the master metric from which it is derived. :return: local_id of master metric, None if not a time comparison metric

gooddata_sdk.insight.InsightService

class gooddata_sdk.insight.**InsightService**(*api_client:* gooddata_sdk.client.GoodDataApiClient)
Bases: object

Insight Service allows retrieval of insights from a GD.CN workspace. The insights are returned as instances of Insight which allows convenient introspection and necessary functions to convert the insight into a form where it can be sent for computation.

Note: the insights are created using GD.CN Analytical Designer or using GoodData.UI SDK. They are stored as visualization objects with a free-form body. This body is specific for AD & SDK. The Insight wrapper exists to take care of these discrepancies.

__init__(*api_client:* gooddata_sdk.client.GoodDataApiClient) → None

Methods

__init__(api_client)

get_insight(workspace_id, insight_id)	Gets a single insight from a workspace.
get_insights(workspace_id)	Gets all insights for a workspace.

get_insight(*workspace_id: str*, *insight_id: str*) → *gooddata_sdk.insight.Insight*
Gets a single insight from a workspace.

Parameters

- **workspace_id** – identifier of workspace to load insight from
- **insight_id** – identifier of the insight

Returns single insight; the insight will contain sideloaded metadata about the entities it references

Return type *Insight*

get_insights(*workspace_id: str*) → list[*Insight*]
Gets all insights for a workspace. The insights will contain side loaded metadata for all execution entities that they reference.

Parameters **workspace_id** – identifier of workspace to load insights from

Returns all available insights, each insight will contain side loaded metadata about the entities it references

gooddata_sdk.sdk

Classes

GoodDataSdk(client)	Top-level class that wraps all the functionality together.

gooddata_sdk.sdk.GoodDataSdk

class gooddata_sdk.sdk.**GoodDataSdk**(*client:* gooddata_sdk.client.GoodDataApiClient)

Bases: `object`

Top-level class that wraps all the functionality together.

__init__(*client:* gooddata_sdk.client.GoodDataApiClient) → None

Take instance of GoodDataApiClient and return new GoodDataSdk instance.

Useful when customized GoodDataApiClient is needed. Usually users should use *GoodDataSdk.create* classmethod.

Methods

__init__(client)	Take instance of GoodDataApiClient and return new GoodDataSdk instance.
create(host_, token_[, extra_user_agent_])	Create common GoodDataApiClient and return new GoodDataSdk instance.

Attributes

`catalog`
`compute`
`insights`
`tables`

classmethod create(*host_: str*, *token_: str*, *extra_user_agent_: Optional[str] = None*, ***custom_headers_: Optional[str]*) → *gooddata_sdk.sdk.GoodDataSdk*

Create common GoodDataApiClient and return new GoodDataSdk instance. Custom headers are filtered. Headers with None value are removed. It simplifies usage because headers can be created directly from optional values.

This is preferred way of creating GoodDataSdk, when no tweaks are needed.

gooddata_sdk.table

Classes

ExecutionTable(response, first_page)	Represents execution result as a table.
TableService(api_client)	The TableService provides a convenient way to drive computations and access the results in a tabular fashion.

gooddata_sdk.table.ExecutionTable

class gooddata_sdk.table.**ExecutionTable**(*response:* gooddata_sdk.compute.ExecutionResponse, *first_page:* gooddata_sdk.compute.ExecutionResult)

Bases: `object`

Represents execution result as a table. This is a convenience wrapper for executions constructed using the following convention:

- all attributes are in the first dimension
- all metrics are in the second dimension
- if the execution is attribute- or metric-less, then there is always single dimension

The mapping to rows is then as follows:

- both attributes + metrics are on the execution = iteration over first dimension; as many rows as total records in the first dimension (paging.total[0])
- just attributes = iteration over just headers in first dimension; as many rows as total records in the first dimension (paging.total[0])
- just metrics = single row, all metrics values returned in one row

__init__(*response:* gooddata_sdk.compute.ExecutionResponse, *first_page:* gooddata_sdk.compute.ExecutionResult) → None

Methods

__init__(response, first_page)	
read_all()	Returns a generator that will be yielding execution result as rows.

Attributes

`attributes`	
column_ids	Returns column identifiers.
column_metadata	Returns mapping of column identifier to definition of either attribute whose elements will be in that column or metric whose value will be calculated in that column.

continues on next page

Table 136 – continued from previous page

metrics

property column_ids: list[str]
Returns column identifiers. Each row will be a mapping of column identifier to column data.

Returns

property column_metadata: dict[str, Union[Attribute, Metric]]
Returns mapping of column identifier to definition of either attribute whose elements will be in that column or metric whose value will be calculated in that column. :return:

read_all() → Generator[dict[str, Any], None, None]
Returns a generator that will be yielding execution result as rows. Each row is a dict() mapping column identifier to value of that column.

Returns generator yielding dict() representing rows of the table

gooddata_sdk.table.TableService

class gooddata_sdk.table.**TableService**(*api_client:* gooddata_sdk.client.GoodDataApiClient)
Bases: object

The TableService provides a convenient way to drive computations and access the results in a tabular fashion.

Compared to the ComputeService, with this one here you do not have to worry about the layout of the result and do not have to have to work with execution response, access the data using paging.

The ExecutionTable returned by the TableService allows you to iterate over the rows of the calculated data.

__init__(*api_client:* gooddata_sdk.client.GoodDataApiClient) → None

Methods

__init__(api_client)
for_insight(workspace_id, insight)
for_items(workspace_id, items[, filters])

gooddata_sdk.type_converter

Functions

build_stores()	Initialize both AttributeConverterStore and DBTypeConverterStore with Convertors.

gooddata_sdk.type_converter.build_stores

gooddata_sdk.type_converter.**build_stores**() → None
Initialize both AttributeConverterStore and DBTypeConverterStore with Convertors.

Classes

AttributeConverterStore()	Store for conversion of attributes
Converter()	Base Converter class.
ConverterRegistryStore()	Class store TypeConverterRegistry instances for each registered type.
DBTypeConverterStore()	Store for conversion of database types
DateConverter()	
DatetimeConverter()	
IntegerConverter()	
StringConverter()	
TypeConverterRegistry(type_name)	Class stores converters for given type with ability to distinguish converters based on sub-type granularity.

gooddata_sdk.type_converter.AttributeConverterStore

class gooddata_sdk.type_converter.**AttributeConverterStore**
Bases: *gooddata_sdk.type_converter.ConverterRegistryStore*

Store for conversion of attributes

__init__()

Methods

__init__()	
find_converter(type_name[, sub_type])	Find Converter for given type and sub type.
register(type_name, class_[, sub_types])	Register Converter instance created from provided Converter class to given type and list of sub types.
reset()	Reset converters setup

classmethod find_converter(*type_name: str*, *sub_type: Optional[str] = None*) → *gooddata_sdk.type_converter.Converter*
Find Converter for given type and sub type. :param type_name: type name :param sub_type: sub type name

classmethod register(*type_name: str*, *class_: Type[*Converter*]*, *sub_types: Optional[list[str]] = None*) → None
Register Converter instance created from provided Converter class to given type and list of sub types. When sub types are not provided, converter is registered as the default one for given type. :param type_name: type

name :param class_: Converter class :param sub_types: list of sub types or None (default type Converter)

classmethod reset() → None
Reset converters setup

gooddata_sdk.type_converter.Converter

class gooddata_sdk.type_converter.**Converter**
Bases: object

Base Converter class. It defines Converter API and implements support for external type conversion. External type conversion provides ability to plug-in conversion function to Converter

__init__()

Methods

__init__()
db_data_type()
set_external_fnc(fnc)
to_external_type(value)
to_type(value)

Attributes

DEFAULT_DB_DATA_TYPE

gooddata_sdk.type_converter.ConverterRegistryStore

class gooddata_sdk.type_converter.**ConverterRegistryStore**
Bases: object

Class store TypeConverterRegistry instances for each registered type. It provides interface to register converters with type and sub-type and to find converter. The class is not meant to be used directly but as base class for child classes

__init__()

Methods

__init__()	

find_converter(type_name[, sub_type])	Find Converter for given type and sub type.
register(type_name, class_[, sub_types])	Register Converter instance created from provided Converter class to given type and list of sub types.
reset()	Reset converters setup

classmethod find_converter(*type_name: str*, *sub_type: Optional[str] = None*) → *gooddata_sdk.type_converter.Converter*

Find Converter for given type and sub type. :param type_name: type name :param sub_type: sub type name

classmethod register(*type_name: str*, *class_: Type[*Converter*]*, *sub_types: Optional[list[str]] = None*) → None

Register Converter instance created from provided Converter class to given type and list of sub types. When sub types are not provided, converter is registered as the default one for given type. :param type_name: type name :param class_: Converter class :param sub_types: list of sub types or None (default type Converter)

classmethod reset() → None

Reset converters setup

gooddata_sdk.type_converter.DBTypeConverterStore

class gooddata_sdk.type_converter.**DBTypeConverterStore**

Bases: *gooddata_sdk.type_converter.ConverterRegistryStore*

Store for conversion of database types

__init__()

Methods

__init__()	

find_converter(type_name[, sub_type])	Find Converter for given type and sub type.
register(type_name, class_[, sub_types])	Register Converter instance created from provided Converter class to given type and list of sub types.
reset()	Reset converters setup

classmethod find_converter(*type_name: str*, *sub_type: Optional[str] = None*) → *gooddata_sdk.type_converter.Converter*

Find Converter for given type and sub type. :param type_name: type name :param sub_type: sub type name

classmethod register(*type_name: str*, *class_: Type[*Converter*]*, *sub_types: Optional[list[str]] = None*) → None

Register Converter instance created from provided Converter class to given type and list of sub types. When sub types are not provided, converter is registered as the default one for given type. :param type_name: type name :param class_: Converter class :param sub_types: list of sub types or None (default type Converter)

classmethod reset() → None
Reset converters setup

gooddata_sdk.type_converter.DateConverter

class gooddata_sdk.type_converter.**DateConverter**
Bases: *gooddata_sdk.type_converter.Converter*

__init__()

Methods

__init__()	
db_data_type()	
set_external_fnc(fnc)	
to_date(value)	Add first month and first date to incomplete iso date string.
to_external_type(value)	
to_type(value)	

Attributes

DEFAULT_DB_DATA_TYPE

classmethod to_date(*value: str*) → datetime.date
Add first month and first date to incomplete iso date string.

```
>>> assert DateConverter.to_date("2021-01") == date(2021, 1, 1)
>>> assert DateConverter.to_date("1992") == date(1992, 1, 1)
```

gooddata_sdk.type_converter.DatetimeConverter

class gooddata_sdk.type_converter.**DatetimeConverter**

Bases: *gooddata_sdk.type_converter.Converter*

__init__()

Methods

__init__()	
db_data_type()	
set_external_fnc(fnc)	
to_datetime(value)	Append minutes to incomplete datetime string.
to_external_type(value)	
to_type(value)	

Attributes

DEFAULT_DB_DATA_TYPE

classmethod to_datetime(*value: str*) → datetime.datetime

Append minutes to incomplete datetime string.

```
>>> from datetime import datetime
>>> assert DatetimeConverter.to_datetime("2021-01-01 02") == datetime(2021, 1,␣
↪1, 2, 0)
>>> assert DatetimeConverter.to_datetime("2021-01-01 12:34") == datetime(2021,␣
↪1, 1, 12, 34)
```

gooddata_sdk.type_converter.IntegerConverter

class gooddata_sdk.type_converter.**IntegerConverter**

Bases: *gooddata_sdk.type_converter.Converter*

__init__()

Methods

__init__()
db_data_type()
set_external_fnc(fnc)
to_external_type(value)
to_type(value)

Attributes

DEFAULT_DB_DATA_TYPE

gooddata_sdk.type_converter.StringConverter

class gooddata_sdk.type_converter.**StringConverter**

Bases: *gooddata_sdk.type_converter.Converter*

__init__()

Methods

__init__()
db_data_type()
set_external_fnc(fnc)
to_external_type(value)
to_type(value)

Attributes

DEFAULT_DB_DATA_TYPE

gooddata_sdk.type_converter.TypeConverterRegistry

class gooddata_sdk.type_converter.**TypeConverterRegistry**(*type_name: str*)

Bases: object

Class stores converters for given type with ability to distinguish converters based on sub-type granularity.

__init__(*type_name: str*)

Initialize instance with type for which instance is going to be responsible :param type_name: type name

Methods

__init__(type_name)	Initialize instance with type for which instance is going to be responsible :param type_name: type name
converter(sub_type)	Find and return converter instance for a given sub-type.
register(converter, sub_type)	Register converter instance for given sub-type (granularity).

converter(*sub_type: Optional[str]*) → *gooddata_sdk.type_converter.Converter*

Find and return converter instance for a given sub-type. Default converter instance is returned if the sub-type is not found or not provided. When a default converter is not registered, ValueError exception is raised. :param sub_type: sub-type name :return: Converter instance

register(*converter:* gooddata_sdk.type_converter.Converter, *sub_type: Optional[str]*) → None

Register converter instance for given sub-type (granularity). If sub-type is not specified, converter is registered as the default one for the whole type. Default converter can be registered only once. :param converter: converter instance :param sub_type: sub-type name

gooddata_sdk.utils

Functions

id_obj_to_key(id_obj)	Given an object containing an id+type pair, this function will return a string key.
load_all_entities(get_page_func[, page_size])	Loads all entities from a paged resource.

gooddata_sdk.utils.id_obj_to_key

gooddata_sdk.utils.**id_obj_to_key**(*id_obj: Union[str,* gooddata_sdk.compute_model.ObjId*, Dict[str, Dict[str, str]], Dict[str, str]]*) → str

Given an object containing an id+type pair, this function will return a string key.

For convenience, this also recognizes the *ref* format used by GoodData.UI SDK. In that format, the id+type are wrapped in 'identifier'.

Parameters **id_obj** – id object

Returns string that can be used as key

gooddata_sdk.utils.load_all_entities

gooddata_sdk.utils.**load_all_entities**(*get_page_func: functools.partial[Any]*, *page_size: int = 500*) → *AllPagedEntities*

Loads all entities from a paged resource. The primary input to this function is a partial function that is setup with all the fixed parameters. Given this the function will get entities page-by-page and merge them into a single 'pseudo-response' containing data and included attributes.

An example usage:

```
>>> import functools
>>> import gooddata_metadata_client as metadata_client
>>> import gooddata_metadata_client.apis as metadata_apis
>>> api = metadata_apis.EntitiesApi(metadata_client.ApiClient())
>>> get_func = functools.partial(api.get_all_entities_visualization_objects, 'some-
↪workspace-id',
>>>                              include=["ALL"], _check_return_type=False)
>>> vis_objects = load_all_entities(get_func)
```

Parameters

- **get_page_func** – an API controller from the metadata client
- **page_size** – optionally specify page length, default is 500

Returns

Classes

AllPagedEntities(data, included)	
SideLoads(objs)	

gooddata_sdk.utils.AllPagedEntities

class gooddata_sdk.utils.**AllPagedEntities**(*data*, *included*)

Bases: tuple

__init__()

Methods

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

Attributes

data	Alias for field number 0
included	Alias for field number 1

count(*value*, */*)
Return number of occurrences of value.

property data
Alias for field number 0

property included
Alias for field number 1

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

gooddata_sdk.utils.SideLoads

class gooddata_sdk.utils.**SideLoads**(*objs: list[Any]*)
Bases: object

__init__(*objs: list[Any]*) → None

Methods

__init__(objs)
all_for_type(obj_type)
find(id_obj)

3.2 Indices and Tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W