GoodData SDK

Release 0.8.0

GoodData Corporation

Jul 14, 2022

CONTENTS:

GoodData Python SDK provides a clean and convenient Python API to interact with GoodData.CN.

At the moment the SDK provides services to inspect and interact with the semantic layer and to consume analytics.

CHAPTER

ONE

INSTALLATION

1.1 Requirements

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

1.2 Installation

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

```
pip install gooddata-sdk
```

1.3 Troubleshooting

- On MacOS, I am getting an error containig following message:

  ```
  (Caused by SSLError(SSLCertVerificationError(1, '[SSL:
  CERTIFICATE_VERIFY_FAILED] certificate verify failed:  unable to get local
  issuer certificate (_ssl.c:1129)'))).
  ```

  This likely caused by Python and it occurs if you have installed Python installed directly from python.org. To mitigate this problem, please install your SSL certificates in *Macintosh HD -> Applications -> Python -> Install Certificates.command**.

CHAPTER

TWO

SERVICES

All services are accessible by class `gooddata_sdk.GoodDataSdk`. The class forms an entry-point to the SDK.

To create an instance of GoodDataSdk:

```
from gooddata_sdk import GoodDataSdk

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)

# Now you can start calling services.
# For example, get a list of all workspaces from my GoodData.CN project
workspaces = sdk.catalog_workspace.list_workspaces()
```

Supported services:

- *Catalog Workspace*: `gooddata_sdk.catalog_workspace`

  Read, update, create and delete workspaces.

- *Catalog Workspace Content*: `gooddata_sdk.catalog_workspace_content`

  Read catalog objects (datasets and metrics) from a workspace.

- *Catalog Data Source*: `gooddata_sdk.catalog_data_source`

  Read, update, create and delete data sources and read their tables.

- *Insights*: `gooddata_sdk.insights`

  Read insights stored in a workspace.

- *Compute*: `gooddata_sdk.compute`

  Drives computation of analytics for GoodData.CN workspaces. Used by higher level services such as the Table service.

- *Table*: `gooddata_sdk.table`

  Compute and read analytics in typical tabular format.

All service-related articles:

2.1 Catalog Workspace Service

The `gooddata_sdk.catalog_workspace` service enables you to perform the following actions on workspaces:

- Get and list existing workspaces
- Update or delete existing workspaces
- Create new workspaces
- Store and restore workspaces from yaml files

The service supports two types of methods:

- Entity methods let you work with workspaces on a high level using simplified *CatalogWorkspace* entities.
- Declarative methods allow you to work with workspaces on a more granular level by fetching entire workspace layouts, including all of their nested objects.

2.1.1 Entity methods

The *gooddata_sdk.catalog_workspace* supports the following entity API calls:

- `get_workspace(workspace_id: str)`

  Returns *CatalogWorkspace*.

  Get an individual workspace.

- `list_workspaces()`

  Returns *List[CatalogWorkspace]*.

  Get a list of all existing workspaces.

- `create_or_update(workspace: CatalogWorkspace)`

  Create a new workspace or overwrite an existing workspace with the same id

- `delete_workspace(workspace_id: str)`

  Delete a workspace

Example Usage

```
from gooddata_sdk import GoodDataSdk

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)

# List workspaces
workspaces = sdk.catalog_workspace.list_workspaces()

print(workspaces)
# [
#   CatalogWorkspace(id=demo, name=Demo),
#   CatalogWorkspace(id=demo_west, name=Demo West),
#   CatalogWorkspace(id=demo_west_california, name=Demo West California)
```

(continues on next page)

(continued from previous page)

```
# ]

# Create new workspace entity locally
my_workspace_object = CatalogWorkspace(id="test_demo", name="Test demo", parent_id="demo
↪")

# Create workspace
sdk.catalog_workspace.create_or_update(my_workspace_object)

# Edit local workspace entity
my_workspace_object.name = "Test"

# Update workspace
sdk.catalog_workspace.create_or_update(my_workspace_object)

# Get workspace
workspace = sdk.catalog_workspace.get_workspace("demo")

print(workspace)
# CatalogWorkspace(id=demo, name=Demo)

# Delete workspace
sdk.catalog_workspace.delete_workspace("demo")
```

2.1.2 Declarative methods

The *gooddata_sdk.catalog_workspace* supports the following declarative API calls:

- `get_declarative_workspace(workspace_id: str)`

  Returns *CatalogDeclarativeWorkspaceModel*.

  Retrieve a workspace layout.

- `put_declarative_workspace(workspace_id: str)`

  Set a workspace layout.

- `get_declarative_workspaces()`

  Returns *CatalogDeclarativeWorkspaces*.

  Retrieve layout of all workspaces and their hierarchy.

- `put_declarative_workspaces(workspace: CatalogDeclarativeWorkspaces)`

  Set layout of all workspaces and their hierarchy.

- `store_declarative_workspaces(layout_root_path: Path = Path.cwd())`

  Store workspaces layouts in directory hierarchy.

```
gooddata_layouts
└── organization_id
        ├── workspaces
        │       ├── workspace_a
        │       │       ├── analytics_model
```

(continues on next page)

(continued from previous page)

```
            │           │           │   ├── analytical_dashboards
            │           │           │   │       └── analytical_dashboard.yaml
            │           │           │   ├── dashboard_plugins
            │           │           │   │       └── dashboard_plugin.yaml
            │           │           │   ├── filter_contexts
            │           │           │   │       └── filter_context.yaml
            │           │           │   ├── metrics
            │           │           │   │       └── metric.yaml
            │           │           │   └── visualization_objects
            │           │           │           └── visualization_object.yaml
            │           │           ├── ldm
            │           │           │   ├── datasets
            │           │           │   │       └── dataset.yaml
            │           │           │   └── date_instances
            │           │           │           └── date_instance.yaml
            │           │           └── workspace_a.yaml
            │           └── workspace_b
            │                   └── ...
            │
            └── workspaces_data_filters
                    ├── filter_1.yaml
                    └── filter_2.yaml
```

- `load_declarative_workspaces(layout_root_path: Path = Path.cwd())`

  Returns *CatalogDeclarativeWorkspaces*.

  Load declarative workspaces layout, which was stored using *store_declarative_workspaces*.

- `load_and_put_declarative_workspaces(layout_root_path: Path = Path.cwd())`

  This method combines *load_declarative_workspaces* and *put_declarative_workspaces* methods to load and set layouts stored using *store_declarative_workspaces*.

Example Usage

```python
from gooddata_sdk import GoodDataSdk
from pathlib import Path

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)

backup_path = Path("workspace_hierarchy_backup.yaml")

# First create a backup of all workspace layout
sdk.catalog_workspace.store_declarative_workspaces(backup_path)

# Get workspace layout
workspace_layout = sdk.catalog_workspace.get_declarative_workspace("demo")

# Modify workspace layout
workspace_layout.ldm.datasets = []
```

(continues on next page)

(continued from previous page)

```
# Update the workspace layout on the server with your changes
workspace_layout.put_declarative_workspace(workspace_layout)

# If something goes wrong, use your backup to restore your workspaces from backup
sdk.catalog_workspace.load_and_put_declarative_workspaces(backup_path)
```

2.2 Catalog Workspace Content Service

The `gooddata_sdk.catalog_workspace_content` service enables you to list catalog all objects from a workspace. These objects include:

- Datasets
- Metrics
- Facts
- Attributes

The service enables read, put, load and store of declarative layout for LDM (logical data model) and analytics model.

The service supports two types of methods:

- Entity methods let you work with workspace content on a high level using simplified entities.
- Declarative methods allow you to work with workspace content on a more granular level by fetching entire workspace content layouts, including all of their nested objects.

2.2.1 Entity methods

The *gooddata_sdk.catalog_workspace_content* supports the following entity API calls:

- `get_full_catalog(workspace_id: str)`

  Returns *CatalogWorkspaceContent*.

  Retrieve all datasets with attributes, facts, and metrics for a workspace.

Example Usage

```
from gooddata_sdk import GoodDataSdk

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)

workspace_id = "demo"

# Read catalog for demo workspace
catalog = sdk.catalog_workspace_content.get_full_catalog(workspace_id)

# Print all dataset in the workspace
```

(continues on next page)

(continued from previous page)

```
for dataset in catalog.datasets:
    print(str(dataset))

# Print all metrics in the workspace
for metric in catalog.metrics:
    print(str(metric))

# Read list of attributes for demo workspace
attributes = sdk.catalog_workspace_content.get_attributes_catalog(workspace_id)

# Read list of facts for demo workspace
facts = sdk.catalog_workspace_content.get_facts_catalog(workspace_id)
```

2.2.2 Declarative methods

The *gooddata_sdk.catalog_workspace_content* supports the following declarative API calls:

Logical Data Model:

- `get_declarative_ldm(workspace_id: str)`

  Returns *CatalogDeclarativeModel*.

  Retrieve a logical model layout. On CatalogDeclarativeModel user can call `modify_mapped_data_source(data_source_mapping: dict)` method, which substitutes data source id in datasets.

- `put_declarative_ldm(workspace_id: str, ldm: CatalogDeclarativeModel, validator: Optional[DataSourceValidator])`

  Put a logical data model into a given workspace. You can pass an additional validator parameter which checks that for every data source id in the logical data model the corresponding data source exists.

- `store_declarative_ldm(workspace_id: str, layout_root_path: Path = Path.cwd())`

  Store logical data model layout in directory hierarchy.

```
gooddata_layouts
└── organization_id
        └── workspaces
                └── workspace_id
                        └── analytics_model
                                └── ldm
                                    ├── datasets
                                    │       └── dataset.yaml
                                    └── date_instances
                                            └── date_instance.yaml
```

- `load_declarative_ldm(workspace_id: str, layout_root_path: Path = Path.cwd())`

  Returns *CatalogDeclarativeModel*.

  Load declarative LDM layout, which was stored using *store_declarative_ldm*.

- `load_and_put_declarative_ldm(workspace_id: str, layout_root_path: Path = Path.cwd(), validator: Optional[DataSourceValidator])`

This method combines *load_declarative_ldm* and *put_declarative_ldm* methods to load and set layouts stored using *store_declarative_ldm*. You can pass an additional validator parameter which checks that for every data source id in the logical data model the corresponding data source exists.

Analytics Model:

- `get_declarative_analytics_model(workspace_id: str)`

  Returns *CatalogDeclarativeAnalytics*.

  Retrieve an analytics model layout.

- `put_declarative_analytics_model(workspace_id: str, analytics_model: CatalogDeclarativeAnalytics)`

  Put an analytics model into a given workspace.

- `store_declarative_analytics_model(workspace_id: str, layout_root_path: Path = Path.cwd())`

  Store declarative analytics model layout in directory hierarchy.

```
gooddata_layouts
└── organization_id
        └── workspaces
                └── workspace_id
                        └── analytics_model
                                ├── analytical_dashboards
                                │       └── analytical_dashboard.yaml
                                ├── dashboard_plugins
                                │       └── dashboard_plugin.yaml
                                ├── filter_contexts
                                │       └── filter_context.yaml
                                ├── metrics
                                │       └── metric.yaml
                                └── visualization_objects
                                        └── visualization_object.yaml
```

- `load_declarative_analytics_model(workspace_id: str, layout_root_path: Path = Path.cwd())`

  Returns *CatalogDeclarativeAnalytics*.

  Load declarative LDM layout, which was stored using *store_declarative_analytics_model*.

- `load_and_put_declarative_analytics_model(workspace_id: str, layout_root_path: Path = Path.cwd())`

  This method combines *load_declarative_analytics_model* and *put_declarative_analytics_model* methods to load and set layouts stored using *store_declarative_analytics_model*.

Example usage:

```python
from gooddata_sdk import GoodDataSdk

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)
```

(continues on next page)

(continued from previous page)

```
# Get ldm object afterward you can modify it
ldm = sdk.catalog_workspace_content.get_declarative_ldm("demo")

# Modify data source id for datasets
ldm.modify_mapped_data_source({"demo-test-ds": "demo-prod-ds"})

# Put ldm object back to server
sdk.catalog_workspace_content.put_declarative_ldm("demo", ldm)

# Get analytics model object afterward you can modify it
analytics_model = sdk.catalog_workspace_content.get_declarative_analytics_model("demo")

# Put analytics model object back to server
sdk.catalog_workspace_content.put_declarative_analytics_model("demo", analytics_model)
```

2.3 Catalog Data Source Service

The `gooddata_sdk.catalog_data_source` service enables you to manage data sources and list their tables. Data source object represents your database, which you integrate with GoodData.CN.

Generally there are two ways how to register data sources:

- The default way works for all data source types: You specify jdbc url, data source type and relevant credentials.
- Customized way for each of the different data source types. You specify custom attributes relevant for your data source and data source type and the url is set in background.

The service supports three types of methods:

- Entity methods let you work with data sources on a high level using simplified *CatalogDataSource* entities.
- Declarative methods allow you to work with data sources on a more granular level by fetching entire workspace layouts, including all of their nested objects.
- Action methods let you perform an execution of some form of computation.

2.3.1 Entity methods

The *gooddata_sdk.catalog_data_source* supports the following entity API calls:

- `create_or_update_data_source(data_source: CatalogDataSource)`

  Create or update data source.

- `list_data_sources()`

  Returns *List[CatalogDataSource]*.

  Lists all data sources.

- `get_data_source(data_source_id: str)`

  Returns *CatalogDataSource*.

  Retrieve data source using data source id.

- `delete_data_source(data_source_id: str)`

Delete data source using data source id.

- `patch_data_source_attributes(data_source_id: str, attributes: dict)`

  Allows you to apply changes to the given data source.

Example Usage

```
from gooddata_sdk import GoodDataSdk

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)

# Create (or update) data source using general interface - can be used for any type of␣
↪data source
# If data source already exists, it is updated
sdk.catalog_data_source.create_or_update_data_source(
    CatalogDataSource(
        id="test",
        name="Test2",
        data_source_type="POSTGRESQL",
        url="jdbc:postgresql://localhost:5432/demo",
        schema="demo",
        credentials=BasicCredentials(
            username="demouser",
            password="demopass",
        ),
        enable_caching=False,
        url_params=[("param", "value")]
    )
)

# Use Postgres specific interface
sdk.catalog_data_source.create_or_update_data_source(
    CatalogDataSourcePostgres(
        id="test",
        name="Test2",
        db_specific_attributes=PostgresAttributes(
            host="localhost", db_name="demo"
        ),
        schema="demo",
        credentials=BasicCredentials(
            username="demouser",
            password="demopass",
        ),
        enable_caching=False,
        url_params=[("param", "value")]
    )
)

# Create Snowflake data source using specialized interface
sdk.catalog_data_source.create_or_update_data_source(
```

(continues on next page)

(continued from previous page)

```
    CatalogDataSourceSnowflake(
        id="test",
        name="Test2",
        db_specific_attributes=SnowflakeAttributes(
            account="mycompany", warehouse="MYWAREHOUSE", db_name="MYDATABASE"
        ),
        schema="demo",
        credentials=BasicCredentials(
            username="demouser",
            password="demopass",
        ),
        enable_caching=False,
        url_params=[("param", "value")]
    )
)

# BigQuery requires path to credentials file, where service account definition is stored
sdk.catalog_data_source.create_or_update_data_source(
    CatalogDataSourceBigQuery(
        id="test",
        name="Test",
        db_specific_attributes=BigQueryAttributes(
            project_id="project_id"
        ),
        schema="demo",
        credentials=TokenCredentialsFromFile(
            file_path=Path("credentials") / "bigquery_service_account.json"
        ),
        enable_caching=True,
        cache_path=["cache_schema"],
        url_params=[("param", "value")]
    )
)

# Look for other CatalogDataSource classes to find your data source type

# List data sources
data_sources = sdk.catalog_data_source.list_data_sources()

# Get single data source
data_sources = sdk.catalog_data_source.get_data_source('ds_id')

# Delete data source
sdk.catalog_data_source.delete_data_source(data_source_id='ds_id')

# Patch data source attribute(s)
sdk.catalog_data_source.patch_data_source_attributes(data_source_id="ds_id", attributes={
↪"name": "Name2"})
```

2.3.2 Declarative methods

The *gooddata_sdk.catalog_data_source* supports the following declarative API calls:

- `get_declarative_data_sources()`

  Returns *CatalogDeclarativeDataSources*.

  Retrieve all data sources, including their related physical model.

- `put_declarative_data_sources(declarative_data_sources: CatalogDeclarativeDataSources, credentials_path: Optional[Path] = None, test_data_sources: bool = False)`

  Set all data sources, including their related physical model.

- `store_declarative_data_sources(layout_root_path: Path = Path.cwd())`

  Store data sources layouts in directory hierarchy.

```
gooddata_layouts
└── organization_id
        └── data_sources
                ├── data_source_a
                │       ├── pdm
                │       │   ├── table_A.yaml
                │       │   └── table_B.yaml
                │       └── data_source_a.yaml
                └── data_source_b
                        └── pdm
                        │   ├── table_X.yaml
                        │   └── table_Y.yaml
                        └── data_source_b.yaml
```

- `load_declarative_data_sources(layout_root_path: Path = Path.cwd())`

  Returns *CatalogDeclarativeDataSources*.

  Load declarative data sources layout, which was stored using *store_declarative_data_sources*.

- `load_and_put_declarative_data_sources(layout_root_path: Path = Path.cwd(), credentials_path: Optional[Path] = None, test_data_sources: bool = False)`

  This method combines *load_declarative_data_sources* and *put_declarative_data_sources* methods to load and set layouts stored using *store_declarative_data_sources*.

Example usage:

```python
from gooddata_sdk import GoodDataSdk
from pathlib import Path

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)

# Get all data sources
ds_objects = sdk.catalog_data_source.get_declarative_data_sources()
```

(continues on next page)

(continued from previous page)

```
print(ds_objects.data_sources[0])
# CatalogDeclarativeDataSource(id=demo-test-ds, type=POSTGRESQL)

# Put data sources with credentials and test data source connection before put
sdk.catalog_data_source.put_declarative_data_sources(data_sources, Path("credentials"),␣
↪True)
```

2.3.3 Action methods

The *gooddata_sdk.catalog_data_source* supports the following action API calls:

- `generate_logical_model(data_source_id: str, generate_ldm_request: CatalogGenerateLdmRequest)`

  Returns *CatalogDeclarativeModel*.

  Generate logical data model for a data source.

- `register_upload_notification(data_source_id: str)`

  Invalidate cache of your computed reports to force your analytics to be recomputed.

- `scan_data_source(data_source_id: str, scan_request: CatalogScanModelRequest = CatalogScanModelRequest(), report_warnings: bool = False)`

  Returns *CatalogScanResultPdm*.

  Scan data source specified by its id and optionally by specified scan request. *CatalogScanResultPdm* contains PDM and warnings. Warnings contain information about columns which were not added to the PDM because their data types are not supported. Additional parameter report_warnings can be passed to suppress or to report warnings. By default warnings are returned but not reported to STDOUT. If you set report_warnings to True, warnings are reported to STDOUT.

- `scan_and_put_pdm(data_source_id: str, scan_request: CatalogScanModelRequest = CatalogScanModelRequest())`

  This method combines *scan_data_source* and *put_declarative_pdm* methods.

- `scan_schemata(data_source_id: str)`

  Returns *list[str]*.

  Returns a list of schemas that exist in the database and can be configured in the data source entity. Data source managers like Dremio or Drill can work with multiple schemas and schema names can be injected into scan_request to filter out tables stored in the different schemas.

Example usage:

```
from gooddata_sdk import GoodDataSdk, CatalogGenerateLdmRequest

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)

# Scan schemata of the data source
schemata = sdk.catalog_data_source.scan_schemata("demo-test-ds")
```

(continues on next page)

(continued from previous page)

```
print(schemata)
# ['demo']

# Scan and put pdm
sdk.catalog_data_source.scan_and_put_pdm("demo-test-ds")

# Define request for generating ldm
generate_ldm_request = CatalogGenerateLdmRequest(separator="__")

# Generate ldm
declarative_model = sdk.catalog_data_source.generate_logical_model("demo-test-ds",␣
↪generate_ldm_request)

# Invalidate cache of your computed reports
sdk.catalog_data_source.register_upload_notification("demo-test-ds")
```

2.4 Insights Service

The `gooddata_sdk.insights` service gives you access to insights stored in a workspace. It can retrieve all the insights from a workspace or one insight based on its name. Insight instance is the input for other services like a Table service

2.4.1 Entity methods

The *gooddata_sdk.insights* supports the following entity API calls:

- `get_insights(workspace_id: str)`

  Returns *list[Insight]*.

  Retrieve a list of Insight objects.

Example usage:

Read all insights in a workspace:

```
from gooddata_sdk import GoodDataSdk

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)

workspace_id = "demo"

# Reads insights from workspace
insights = sdk.insights.get_insights(workspace_id)
# Print all fetched insights
for insight in insights:
    print(str(insight))
```

2.5 Compute Service

The `gooddata_sdk.compute` service drives computation of analytics for 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.

Higher level services like Table service use Compute service to execute computation in GoodData.CN. Higher level service is also responsible for results presentation to the user e.g. in tabular form.

2.5.1 Entity methods

The *gooddata_sdk.compute* supports the following entity API calls:

- `for_exec_def(workspace_id: str, exec_def: ExecutionDefinition)`

  Returns *ExecutionResponse*.

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

2.6 Table Service

The `gooddata_sdk.table` service allows you to consume analytics in typical tabular format. The service allows free-form computations and computations of data for GoodData.CN Insights.

2.6.1 Entity methods

The *gooddata_sdk.table* supports the following entity API calls:

- `for_insight(workspace_id: str, insight: Insight)`

  Returns *ExecutionTable*.

  Retrieve data as an ExecutionTable from the given insight.

- `for_items(workspace_id: str, items: list[Union[Attribute, Metric]], filters: Optional[list[Filter]] = None)`

  Returns *ExecutionTable*.

  Retrieve data as an ExecutionTable from the given list of attributes/metrics, and filters.

Example usage:

Get tabular data for an insight defined on your GoodData.CN server:

```
from gooddata_sdk import GoodDataSdk

# GoodData.CN host in the form of uri eg. "http://localhost:3000"
host = "http://localhost:3000"
# GoodData.CN user token
token = "some_user_token"
sdk = GoodDataSdk.create(host, token)

workspace_id = "demo"
insight_id = "some_insight_id_in_demo_workspace"
```

(continues on next page)

(continued from previous page)

```
# Reads insight from workspace
insight = sdk.insights.get_insight(workspace_id, insight_id)

# Triggers computation for the insight. the result will be returned in a tabular form
table = sdk.tables.for_insight(workspace_id, insight)

# This is how you can read data row-by-row and do something with it
for row in table.read_all():
    print(row)

# An example of data printed for insight top_10_products
# {'781952e728204dcf923142910cc22ae2': 'Biolid', 'fe513cef1c6244a5ac21c5f49c56b108': 'Outdoor
↪', '77dc71bbac92412bac5f94284a5919df': 34697.71}
# {'781952e728204dcf923142910cc22ae2': 'ChalkTalk', 'fe513cef1c6244a5ac21c5f49c56b108': 'Home
↪', '77dc71bbac92412bac5f94284a5919df': 17657.35}
# {'781952e728204dcf923142910cc22ae2': 'Elentrix', 'fe513cef1c6244a5ac21c5f49c56b108':
↪'Outdoor', '77dc71bbac92412bac5f94284a5919df': 27662.09}
# {'781952e728204dcf923142910cc22ae2': 'Integres', 'fe513cef1c6244a5ac21c5f49c56b108':
↪'Outdoor', '77dc71bbac92412bac5f94284a5919df': 47766.74}
# {'781952e728204dcf923142910cc22ae2': 'Magnemo', 'fe513cef1c6244a5ac21c5f49c56b108':
↪'Electronics', '77dc71bbac92412bac5f94284a5919df': 44026.52}
# {'781952e728204dcf923142910cc22ae2': 'Neptide', 'fe513cef1c6244a5ac21c5f49c56b108': 'Outdoor
↪', '77dc71bbac92412bac5f94284a5919df': 99440.44}
# {'781952e728204dcf923142910cc22ae2': 'Optique', 'fe513cef1c6244a5ac21c5f49c56b108': 'Home',
↪'77dc71bbac92412bac5f94284a5919df': 40307.76}
# {'781952e728204dcf923142910cc22ae2': 'PortaCode', 'fe513cef1c6244a5ac21c5f49c56b108':
↪'Electronics', '77dc71bbac92412bac5f94284a5919df': 18841.17}
# {'781952e728204dcf923142910cc22ae2': 'Slacks', 'fe513cef1c6244a5ac21c5f49c56b108': 'Clothing
↪', '77dc71bbac92412bac5f94284a5919df': 18469.15}
# {'781952e728204dcf923142910cc22ae2': 'T-Shirt', 'fe513cef1c6244a5ac21c5f49c56b108':
↪'Clothing', '77dc71bbac92412bac5f94284a5919df': 17937.49}
```

CHAPTER

THREE

API REFERENCE

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

3.1 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.insight	
gooddata_sdk.sdk	
gooddata_sdk.support	
gooddata_sdk.table	
gooddata_sdk.type_converter	
gooddata_sdk.utils	

3.1.1 gooddata_sdk.catalog

Modules

gooddata_sdk.catalog.catalog_service_base

Classes

gooddata_sdk.catalog.catalog_service_base.CatalogServiceBase

class gooddata_sdk.catalog.catalog_service_base.**CatalogServiceBase**(*api_client:* gooddata_sdk.client.GoodDataApiClient)

Bases: object

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

Methods

Attributes

organization_id

gooddata_sdk.catalog.data_source

Modules

gooddata_sdk.catalog.data_source.action_requests
gooddata_sdk.catalog.data_source.declarative_model
gooddata_sdk.catalog.data_source.entity_model
gooddata_sdk.catalog.data_source.service
gooddata_sdk.catalog.data_source.validation

gooddata_sdk.catalog.data_source.action_requests

Modules

gooddata_sdk.catalog.data_source.action_requests.ldm_request
gooddata_sdk.catalog.data_source.action_requests.scan_model_request

gooddata_sdk.catalog.data_source.action_requests.ldm_request

Classes

CatalogGenerateLdmRequest(separator[, ...])

gooddata_sdk.catalog.data_source.action_requests.ldm_request.CatalogGenerateLdmRequest

class gooddata_sdk.catalog.data_source.action_requests.ldm_request.**CatalogGenerateLdmRequest**(*separator: str*, *generate_long_ids: Optional[bool] = None*, *table_prefix: Optional[str] = None*, *view_prefix: Optional[str] = None*, *primary_label_p Optional[str] = None*, *secondary_label Optional[str] = None*, *fact_prefix: Optional[str] = None*, *date_granula Optional[str] = None*, *grain_prefix: Optional[str] = None*, *reference_prefix: Optional[str] = None*, *grain_referen Optional[str] = None*

__init__(*separator: str*, *generate_long_ids: Optional[bool] = None*, *table_prefix: Optional[str] = None*, *view_prefix: Optional[str] = None*, *primary_label_prefix: Optional[str] = None*, *secondary_label_prefix: Optional[str] = None*, *fact_prefix: Optional[str] = None*, *date_granularities: Optional[str] = None*, *grain_prefix: Optional[str] = None*, *reference_prefix: Optional[str] = None*, *grain_reference_prefix: Optional[str] = None*, *denorm_prefix: Optional[str] = None*, *wdf_prefix: Optional[str] = None*)

Methods

__init__(separator[, generate_long_ids, ...])
to_api()

gooddata_sdk.catalog.data_source.action_requests.scan_model_request

Classes

CatalogScanModelRequest([separator, ...])

gooddata_sdk.catalog.data_source.action_requests.scan_model_request.CatalogScanModelRequest

class gooddata_sdk.catalog.data_source.action_requests.scan_model_request.**CatalogScanModelRequest**(*separa*
str
=
'__',
scan_t
bool
=
True,
scan_v
bool
=
False,
ta-
ble_pre
Op-
tional[
=
None,
view_p
Op-
tional[
=
None)

Bases: object

__init__(*separator: str = '__'*, *scan_tables: bool = True*, *scan_views: bool = False*, *table_prefix: Optional[str] = None*, *view_prefix: Optional[str] = None*)

Methods

__init__([separator, scan_tables, ...])
to_api()

gooddata_sdk.catalog.data_source.declarative_model

Modules

gooddata_sdk.catalog.data_source.declarative_model.data_source
gooddata_sdk.catalog.data_source.declarative_model.physical_model

gooddata_sdk.catalog.data_source.declarative_model.data_source

Classes

CatalogDeclarativeDataSource(id, type, name, ...)
CatalogDeclarativeDataSources(data_sources)

gooddata_sdk.catalog.data_source.declarative_model.data_source.CatalogDeclarativeDataSource

class gooddata_sdk.catalog.data_source.declarative_model.data_source.**CatalogDeclarativeDataSource**(*id: str*, *type: str*, *name: str*, *url: str*, *schema: str*, *enable_c Optional[pdm: Optional[cache_ Optional[= None*, *username: Optional[= None*, *permissions: list[Ca = None*)

Bases: *gooddata_sdk.catalog.entity.CatalogTypeEntity*

__init__(*id: str*, *type: str*, *name: str*, *url: str*, *schema: str*, *enable_caching: Optional[bool]*, *pdm: Optional[CatalogDeclarativeTables]*, *cache_path: Optional[list[str]] = None*, *username: Optional[str] = None*, *permissions: list[CatalogDeclarativeDataSourcePermission] = None*)

Methods

__init__(id, type, name, url, schema, ...[, ...])	
data_source_folder(data_sources_folder, ...)	
from_api(entity)	
load_from_disk(data_sources_folder, ...)	

continues on next page

Table 15 – continued from previous page

store_to_disk(data_sources_folder)
to_api([password, token, ...])
to_test_request([password, token])

gooddata_sdk.catalog.data_source.declarative_model.data_source.CatalogDeclarativeDataSources

class gooddata_sdk.catalog.data_source.declarative_model.data_source.**CatalogDeclarativeDataSources**(*data_ list[C*

Bases: object

__init__(*data_sources: list[*CatalogDeclarativeDataSource*]*)

Methods

__init__(data_sources)
data_sources_folder(layout_organization_folder)
from_api(entity)
from_dict(data[, camel_case]) **param data** Data loaded for example from the file.
load_from_disk(layout_organization_folder)
store_to_disk(layout_organization_folder)
to_api([credentials])

classmethod from_dict(*data: dict[str, Any]*, *camel_case: bool = True*) → *CatalogDeclarativeDataSources*

Parameters

- **data** – Data loaded for example from the file.
- **camel_case** – True if the variable names in the input data are serialized names as specified in the OpenAPI document. False if the variables names in the input data are python variable names in PEP-8 snake case.

Returns CatalogDeclarativeDataSources object.

gooddata_sdk.catalog.data_source.declarative_model.physical_model

Modules

gooddata_sdk.catalog.data_source.declarative_model.physical_model.column
gooddata_sdk.catalog.data_source.declarative_model.physical_model.pdm
gooddata_sdk.catalog.data_source.declarative_model.physical_model.table

gooddata_sdk.catalog.data_source.declarative_model.physical_model.column

Classes

CatalogDeclarativeColumn(name, data_type, ...)

gooddata_sdk.catalog.data_source.declarative_model.physical_model.column.CatalogDeclarativeColumn

class gooddata_sdk.catalog.data_source.declarative_model.physical_model.column.**CatalogDeclarativeColumn**

Bases: `object`

__init__(*name: str*, *data_type: str*, *is_primary_key: Optional[bool]*, *referenced_table_id: Optional[str]*, *referenced_table_column: Optional[str]*)

Methods

__init__(name, data_type, is_primary_key, ...)
from_api(entity)
to_api()

gooddata_sdk.catalog.data_source.declarative_model.physical_model.pdm

Functions

get_pdm_folder(data_source_folder)

gooddata_sdk.catalog.data_source.declarative_model.physical_model.pdm.get_pdm_folder

gooddata_sdk.catalog.data_source.declarative_model.physical_model.pdm.**get_pdm_folder**(*data_source_folder: pathlib.Path*) → pathlib.Path

Classes

CatalogDeclarativeTables(tables)
CatalogScanResultPdm(pdm, warnings)

gooddata_sdk.catalog.data_source.declarative_model.physical_model.pdm.CatalogDeclarativeTables

class gooddata_sdk.catalog.data_source.declarative_model.physical_model.pdm.**CatalogDeclarativeTables**(*tab list*

Bases: object

__init__(*tables: list[CatalogDeclarativeTable]*)

Methods

__init__(tables)	
from_api(entity)	
from_dict(data[, camel_case])	**param data** Data loaded for example from the file.
load_from_disk(data_source_folder)	
store_to_disk(data_source_folder)	
to_api()	

classmethod from_dict(*data: dict[str, Any]*, *camel_case: bool = True*) → *CatalogDeclarativeTables*

Parameters

- **data** – Data loaded for example from the file.
- **camel_case** – True if the variable names in the input data are serialized names as specified in the OpenAPI document. False if the variables names in the input data are python variable names in PEP-8 snake case.

Returns DeclarativeTables object.

gooddata_sdk.catalog.data_source.declarative_model.physical_model.pdm.CatalogScanResultPdm

class gooddata_sdk.catalog.data_source.declarative_model.physical_model.pdm.**CatalogScanResultPdm**(*pdm:* CatalogDeclarativeTables, *warnings: list[dict]*)

Bases: object

__init__(*pdm:* CatalogDeclarativeTables, *warnings: list[dict]*)

Methods

__init__(pdm, warnings)
from_api(entity)

gooddata_sdk.catalog.data_source.declarative_model.physical_model.table

Classes

CatalogDeclarativeTable(id, type, path, columns)

gooddata_sdk.catalog.data_source.declarative_model.physical_model.table.CatalogDeclarativeTable

class gooddata_sdk.catalog.data_source.declarative_model.physical_model.table.**CatalogDeclarativeTable**(*id*, *type*, *path*, *columns*)

Bases: *gooddata_sdk.catalog.entity.CatalogTypeEntity*

__init__(*id: str*, *type: str*, *path: list[str]*, *columns: list[CatalogDeclarativeColumn]*)

Methods

__init__(id, type, path, columns)
from_api(entity)
store_to_disk(pdm_folder)
to_api()

gooddata_sdk.catalog.data_source.entity_model

Modules

gooddata_sdk.catalog.data_source.entity_model.content_objects
gooddata_sdk.catalog.data_source.entity_model.data_source

gooddata_sdk.catalog.data_source.entity_model.content_objects

Modules

gooddata_sdk.catalog.data_source.entity_model.content_objects.table

gooddata_sdk.catalog.data_source.entity_model.content_objects.table

Classes

CatalogDataSourceTable(entity)
CatalogDataSourceTableColumn(column)

gooddata_sdk.catalog.data_source.entity_model.content_objects.table.CatalogDataSourceTable

class gooddata_sdk.catalog.data_source.entity_model.content_objects.table.**CatalogDataSourceTable**(*entity: dict[str, Any]*)

Bases: *gooddata_sdk.catalog.entity.CatalogEntity*

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

Methods

__init__(entity)

Attributes

columns
description
id
obj_id
path
table_type
title
type
username

gooddata_sdk.catalog.data_source.entity_model.content_objects.table.CatalogDataSourceTableColumn

class gooddata_sdk.catalog.data_source.entity_model.content_objects.table.**CatalogDataSourceTableColumn**(

Bases: object

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

Methods

__init__(column)

Attributes

data_type
name
primary_key
referenced_table_column
referenced_table_id

gooddata_sdk.catalog.data_source.entity_model.data_source

Classes

BigQueryAttributes(project_id[, port])	
CatalogDataSource(id, name, schema, credentials)	
CatalogDataSourceBigQuery(id, name, schema, ...)	
CatalogDataSourcePostgres(id, name, schema, ...)	
CatalogDataSourceRedshift(id, name, schema, ...)	
CatalogDataSourceSnowflake(id, name, schema, ...)	
CatalogDataSourceVertica(id, name, schema, ...)	
DatabaseAttributes()	
PostgresAttributes(host, db_name[, port])	
RedshiftAttributes(host, db_name[, port])	
SnowflakeAttributes(account, warehouse, db_name)	
VerticaAttributes(host, db_name[, port])	

gooddata_sdk.catalog.data_source.entity_model.data_source.BigQueryAttributes

class gooddata_sdk.catalog.data_source.entity_model.data_source.**BigQueryAttributes**(*project_id: str*, *port: str = '443'*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.DatabaseAttributes*

__init__(*project_id: str*, *port: str = '443'*)

Methods

__init__(project_id[, port])	

Attributes

str_attributes

gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSource

class gooddata_sdk.catalog.data_source.entity_model.data_source.**CatalogDataSource**(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Bases: *gooddata_sdk.catalog.entity.CatalogNameEntity*

__init__(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Methods

`__init__`(id, name, schema, credentials[, ...])	
`from_api`(entity)	
`to_api`()	
`to_api_patch`(data_source_id, attributes)	

gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSourceBigQuery

class gooddata_sdk.catalog.data_source.entity_model.data_source.**CatalogDataSourceBigQuery**(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSource*

__init__(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Methods

`__init__`(id, name, schema, credentials[, ...])	
`from_api`(entity)	
`to_api`()	
`to_api_patch`(data_source_id, attributes)	

gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSourcePostgres

class gooddata_sdk.catalog.data_source.entity_model.data_source.**CatalogDataSourcePostgres**(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSource*

__init__(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Methods

`__init__`(id, name, schema, credentials[, ...])
`from_api`(entity)
`to_api`()
`to_api_patch`(data_source_id, attributes)

gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSourceRedshift

class gooddata_sdk.catalog.data_source.entity_model.data_source.**CatalogDataSourceRedshift**(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSourcePostgres*

__init__(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Methods

`__init__`(id, name, schema, credentials[, ...])	
`from_api`(entity)	
`to_api`()	
`to_api_patch`(data_source_id, attributes)	

gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSourceSnowflake

class gooddata_sdk.catalog.data_source.entity_model.data_source.**CatalogDataSourceSnowflake**(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSource*

__init__(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Methods

`__init__`(id, name, schema, credentials[, ...])
`from_api`(entity)
`to_api`()
`to_api_patch`(data_source_id, attributes)

gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSourceVertica

class gooddata_sdk.catalog.data_source.entity_model.data_source.**CatalogDataSourceVertica**(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attribu Optional[DatabaseAtt = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[s str]]] = None*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.CatalogDataSourcePostgres*

__init__(*id: str*, *name: str*, *schema: str*, *credentials: Credentials*, *url: Optional[str] = None*, *data_source_type: Optional[str] = None*, *db_specific_attributes: Optional[DatabaseAttributes] = None*, *enable_caching: Optional[bool] = None*, *cache_path: Optional[list[str]] = None*, *url_params: Optional[List[Tuple[str, str]]] = None*)

Methods

__init__(id, name, schema, credentials[, ...])
from_api(entity)
to_api()
to_api_patch(data_source_id, attributes)

gooddata_sdk.catalog.data_source.entity_model.data_source.DatabaseAttributes

class gooddata_sdk.catalog.data_source.entity_model.data_source.**DatabaseAttributes**

Bases: object

__init__()

Methods

__init__()

Attributes

str_attributes

gooddata_sdk.catalog.data_source.entity_model.data_source.PostgresAttributes

class gooddata_sdk.catalog.data_source.entity_model.data_source.**PostgresAttributes**(*host: str*, *db_name: str*, *port: str = '5432'*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.DatabaseAttributes*

__init__(*host: str*, *db_name: str*, *port: str = '5432'*)

Methods

__init__(host, db_name[, port])

Attributes

str_attributes

gooddata_sdk.catalog.data_source.entity_model.data_source.RedshiftAttributes

class gooddata_sdk.catalog.data_source.entity_model.data_source.**RedshiftAttributes**(*host: str*, *db_name: str*, *port: str = '5439'*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.PostgresAttributes*

__init__(*host: str*, *db_name: str*, *port: str = '5439'*)

Methods

__init__(host, db_name[, port])

Attributes

str_attributes

gooddata_sdk.catalog.data_source.entity_model.data_source.SnowflakeAttributes

class gooddata_sdk.catalog.data_source.entity_model.data_source.**SnowflakeAttributes**(*account: str*, *warehouse: str*, *db_name: str*, *port: str = '443'*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.DatabaseAttributes*

__init__(*account: str*, *warehouse: str*, *db_name: str*, *port: str = '443'*)

Methods

__init__(account, warehouse, db_name[, port])	

Attributes

str_attributes

gooddata_sdk.catalog.data_source.entity_model.data_source.VerticaAttributes

class gooddata_sdk.catalog.data_source.entity_model.data_source.**VerticaAttributes**(*host: str*, *db_name: str*, *port: str = '5433'*)

Bases: *gooddata_sdk.catalog.data_source.entity_model.data_source.PostgresAttributes*

__init__(*host: str*, *db_name: str*, *port: str = '5433'*)

Methods

__init__(host, db_name[, port])	

Attributes

str_attributes

gooddata_sdk.catalog.data_source.service

Classes

CatalogDataSourceService(api_client)	

gooddata_sdk.catalog.data_source.service.CatalogDataSourceService

class gooddata_sdk.catalog.data_source.service.**CatalogDataSourceService**(*api_client:* gooddata_sdk.client.GoodDataApiClient)

Bases: *gooddata_sdk.catalog.catalog_service_base.CatalogServiceBase*

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

Methods

__init__(api_client)
create_or_update_data_source(data_source)
data_source_folder(data_source_id, ...)
delete_data_source(data_source_id)
generate_logical_model(data_source_id, ...)
get_data_source(data_source_id)
get_declarative_data_sources()
get_declarative_pdm(data_source_id)
get_organization()
layout_organization_folder(layout_root_path)
list_data_source_tables(data_source_id)
list_data_sources()
load_and_put_declarative_data_sources([...])
load_and_put_declarative_pdm(data_source_id)
load_declarative_data_sources([layout_root_path])
load_declarative_pdm(data_source_id[, ...])
patch_data_source_attributes(data_source_id, ...)
put_declarative_data_sources(...[, ...])
put_declarative_pdm(data_source_id, ...)
register_upload_notification(data_source_id)

continues on next page

Table 53 – continued from previous page

report_warnings(warnings)	
scan_and_put_pdm(data_source_id[, scan_request])	
scan_data_source(data_source_id[, ...])	
scan_schemata(data_source_id)	
store_declarative_data_sources([...])	
store_declarative_pdm(data_source_id[, ...])	
test_data_sources_connection(...[, ...])	

Attributes

organization_id

gooddata_sdk.catalog.data_source.validation

Modules

gooddata_sdk.catalog.data_source.validation.data_source

gooddata_sdk.catalog.data_source.validation.data_source

Classes

DataSourceValidator(data_source_service)

gooddata_sdk.catalog.data_source.validation.data_source.DataSourceValidator

class gooddata_sdk.catalog.data_source.validation.data_source.**DataSourceValidator**(*data_source_service:* good-data_sdk.catalog.data_sourc

Bases: object

__init__(*data_source_service:* gooddata_sdk.catalog.data_source.service.CatalogDataSourceService)

Methods

__init__(data_source_service)
validate_data_source_ids(data_source_ids)
validate_ldm(model)

gooddata_sdk.catalog.entity

Classes

BasicCredentials(username, password)
CatalogEntity(entity)
CatalogNameEntity(id, name)
CatalogTitleEntity(id, title)
CatalogTypeEntity(id, type)
Credentials()
TokenCredentials(token)
TokenCredentialsFromFile(file_path)

gooddata_sdk.catalog.entity.BasicCredentials

class gooddata_sdk.catalog.entity.**BasicCredentials**(*username: str*, *password: str*)

Bases: *gooddata_sdk.catalog.entity.Credentials*

__init__(*username: str*, *password: str*)

Methods

__init__(username, password)
create(creds_classes, entity)
from_api(attributes)
is_part_of_api(entity)

continues on next page

Table 59 – continued from previous page

to_api_args()
validate_instance(creds_classes, instance)

Attributes

PASSWORD_KEY
USER_KEY

gooddata_sdk.catalog.entity.CatalogEntity

class gooddata_sdk.catalog.entity.**CatalogEntity**(*entity: dict[str, Any]*)

Bases: object

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

Methods

__init__(entity)

Attributes

description
id
obj_id
title
type

gooddata_sdk.catalog.entity.CatalogNameEntity

class gooddata_sdk.catalog.entity.**CatalogNameEntity**(*id: str*, *name: str*)
Bases: object

__init__(*id: str*, *name: str*)

Methods

__init__(id, name)

gooddata_sdk.catalog.entity.CatalogTitleEntity

class gooddata_sdk.catalog.entity.**CatalogTitleEntity**(*id: str*, *title: str*)
Bases: object

__init__(*id: str*, *title: str*)

Methods

__init__(id, title)
from_api(entity)

gooddata_sdk.catalog.entity.CatalogTypeEntity

class gooddata_sdk.catalog.entity.**CatalogTypeEntity**(*id: str*, *type: str*)
Bases: object

__init__(*id: str*, *type: str*)

Methods

__init__(id, type)
from_api(entity)

gooddata_sdk.catalog.entity.Credentials

class gooddata_sdk.catalog.entity.**Credentials**

Bases: object

__init__()

Methods

__init__()
create(creds_classes, entity)
from_api(entity)
is_part_of_api(entity)
to_api_args()
validate_instance(creds_classes, instance)

gooddata_sdk.catalog.entity.TokenCredentials

class gooddata_sdk.catalog.entity.**TokenCredentials**(*token: str*)

Bases: *gooddata_sdk.catalog.entity.Credentials*

__init__(*token: str*)

Methods

__init__(token)
create(creds_classes, entity)
from_api(entity)
is_part_of_api(entity)
to_api_args()
validate_instance(creds_classes, instance)

Attributes

TOKEN_KEY
USER_KEY

gooddata_sdk.catalog.entity.TokenCredentialsFromFile

class gooddata_sdk.catalog.entity.**TokenCredentialsFromFile**(*file_path: pathlib.Path*)

Bases: *gooddata_sdk.catalog.entity.Credentials*

__init__(*file_path: pathlib.Path*)

Methods

__init__(file_path)
create(creds_classes, entity)
from_api(entity)
is_part_of_api(entity)
to_api_args()
token_from_file(file_path)
validate_instance(creds_classes, instance)

Attributes

TOKEN_KEY
USER_KEY

gooddata_sdk.catalog.identifier

Classes

CatalogAssigneeIdentifier(id, type)
CatalogGrainIdentifier(id, type)

continues on next page

Table 71 – continued from previous page

CatalogIdentifierBase(id)	
CatalogReferenceIdentifier(id)	
CatalogWorkspaceIdentifier(id)	

gooddata_sdk.catalog.identifier.CatalogAssigneeIdentifier

class gooddata_sdk.catalog.identifier.**CatalogAssigneeIdentifier**(*id: str*, *type: str*)

Bases: *gooddata_sdk.catalog.entity.CatalogTypeEntity*

__init__(*id: str*, *type: str*)

Methods

__init__(id, type)	
from_api(entity)	
to_api()	

gooddata_sdk.catalog.identifier.CatalogGrainIdentifier

class gooddata_sdk.catalog.identifier.**CatalogGrainIdentifier**(*id: str*, *type: str*)

Bases: *gooddata_sdk.catalog.entity.CatalogTypeEntity*

__init__(*id: str*, *type: str*)

Methods

__init__(id, type)	
from_api(entity)	
to_api()	

gooddata_sdk.catalog.identifier.CatalogIdentifierBase

class gooddata_sdk.catalog.identifier.**CatalogIdentifierBase**(*id: str*)
Bases: object

__init__(*id: str*)

Methods

__init__(id)
from_api(entity)

gooddata_sdk.catalog.identifier.CatalogReferenceIdentifier

class gooddata_sdk.catalog.identifier.**CatalogReferenceIdentifier**(*id: str*)
Bases: *gooddata_sdk.catalog.identifier.CatalogIdentifierBase*

__init__(*id: str*)

Methods

__init__(id)
from_api(entity)
to_api()

gooddata_sdk.catalog.identifier.CatalogWorkspaceIdentifier

class gooddata_sdk.catalog.identifier.**CatalogWorkspaceIdentifier**(*id: str*)
Bases: *gooddata_sdk.catalog.identifier.CatalogIdentifierBase*

__init__(*id: str*)

Methods

__init__(id)
from_api(entity)
to_api()

gooddata_sdk.catalog.organization

Modules

gooddata_sdk.catalog.organization.entity_model
gooddata_sdk.catalog.organization.service

gooddata_sdk.catalog.organization.entity_model

Modules

gooddata_sdk.catalog.organization.entity_model.organization

gooddata_sdk.catalog.organization.entity_model.organization

Classes

CatalogOrganization(organization_id, name, ...)

gooddata_sdk.catalog.organization.entity_model.organization.CatalogOrganization

class gooddata_sdk.catalog.organization.entity_model.organization.**CatalogOrganization**(*organization_id: str*, *name: str*, *hostname: str*)

Bases: *gooddata_sdk.catalog.entity.CatalogNameEntity*

__init__(*organization_id: str*, *name: str*, *hostname: str*) → None

Methods

__init__(organization_id, name, hostname)
from_api(entity)
to_api()

gooddata_sdk.catalog.organization.service

Classes

CatalogOrganizationService(api_client)	

gooddata_sdk.catalog.organization.service.CatalogOrganizationService

class gooddata_sdk.catalog.organization.service.**CatalogOrganizationService**(*api_client:* gooddata_sdk.client.GoodDataApiClient)

Bases: *gooddata_sdk.catalog.catalog_service_base.CatalogServiceBase*

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

Methods

__init__(api_client)	
get_organization()	
layout_organization_folder(layout_root_path)	

Attributes

organization_id	

gooddata_sdk.catalog.permissions

Modules

gooddata_sdk.catalog.permissions.permission	

gooddata_sdk.catalog.permissions.permission

Classes

CatalogDeclarativeDataSourcePermission(name, ...)	
CatalogDeclarativeSingleWorkspacePermission(...)	

continues on next page

Table 85 – continued from previous page

CatalogDeclarativeWorkspaceHierarchyPermission(...)	
CatalogDeclarativeWorkspacePermissions([...])	
PermissionBase(name, assignee)	

gooddata_sdk.catalog.permissions.permission.CatalogDeclarativeDataSourcePermission

class gooddata_sdk.catalog.permissions.permission.**CatalogDeclarativeDataSourcePermission**(*name: str*, *assignee:* gooddata_sdk.catalog.id

Bases: *gooddata_sdk.catalog.permissions.permission.PermissionBase*

__init__(*name: str*, *assignee:* gooddata_sdk.catalog.identifier.CatalogAssigneeIdentifier)

Methods

__init__(name, assignee)	
from_api(entity)	
to_api()	

gooddata_sdk.catalog.permissions.permission.CatalogDeclarativeSingleWorkspacePermission

class gooddata_sdk.catalog.permissions.permission.**CatalogDeclarativeSingleWorkspacePermission**(*name: str*, *assignee:* gooddata_sdk.cat

Bases: *gooddata_sdk.catalog.permissions.permission.PermissionBase*

__init__(*name: str*, *assignee:* gooddata_sdk.catalog.identifier.CatalogAssigneeIdentifier)

Methods

__init__(name, assignee)
from_api(entity)
to_api()

gooddata_sdk.catalog.permissions.permission.CatalogDeclarativeWorkspaceHierarchyPermission

class gooddata_sdk.catalog.permissions.permission.**CatalogDeclarativeWorkspaceHierarchyPermission**(*name: str*, *assignee:* gooddata_sdk

Bases: *gooddata_sdk.catalog.permissions.permission.PermissionBase*

__init__(*name: str*, *assignee:* gooddata_sdk.catalog.identifier.CatalogAssigneeIdentifier)

Methods

__init__(name, assignee)
from_api(entity)
to_api()

gooddata_sdk.catalog.permissions.permission.CatalogDeclarativeWorkspacePermissions

class gooddata_sdk.catalog.permissions.permission.**CatalogDeclarativeWorkspacePermissions**(*permissions: list[*CatalogDeclara = *None*, *hierarchy_permissions: list[*CatalogDeclara = *None*)

Bases: object

__init__(*permissions: list[*CatalogDeclarativeSingleWorkspacePermission*] = None*, *hierarchy_permissions: list[*CatalogDeclarativeWorkspaceHierarchyPermission*] = None*)

Methods

__init__([permissions, hierarchy_permissions])
from_api(entity)
to_api()

gooddata_sdk.catalog.permissions.permission.PermissionBase

class gooddata_sdk.catalog.permissions.permission.**PermissionBase**(*name: str*, *assignee:* gooddata_sdk.catalog.identifier.CatalogAssigneeIdentif

Bases: object

__init__(*name: str*, *assignee:* gooddata_sdk.catalog.identifier.CatalogAssigneeIdentifier)

Methods

__init__(name, assignee)
from_api(entity)

gooddata_sdk.catalog.types

gooddata_sdk.catalog.workspace

Modules

gooddata_sdk.catalog.workspace.declarative_model
gooddata_sdk.catalog.workspace.entity_model
gooddata_sdk.catalog.workspace.model_container
gooddata_sdk.catalog.workspace.service

gooddata_sdk.catalog.workspace.declarative_model

Modules

gooddata_sdk.catalog.workspace.declarative_model.workspace

gooddata_sdk.catalog.workspace.declarative_model.workspace

Modules

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model
gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model
gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model

Modules

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model

Classes

CatalogAnalyticsBase(id, title, content[, ...])
CatalogDeclarativeAnalyticalDashboard(id, ...)
CatalogDeclarativeAnalytics([analytics])
CatalogDeclarativeAnalyticsLayer([...])
CatalogDeclarativeDashboardPlugin(id, title, ...)
CatalogDeclarativeFilterContext(id, title, ...)
CatalogDeclarativeMetric(id, title, content)

continues on next page

Table 95 – continued from previous page

CatalogDeclarativeVisualizationObject(id, ...)	

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogAnalytic

class gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.**Catalog**

Bases: `object`

__init__(*id: str*, *title: str*, *content: dict[str, Any]*, *description: str = None*, *tags: list[str] = None*)

Methods

__init__(id, title, content[, description, tags])	
from_api(entity)	
from_dict(data)	For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case.
get_kwargs()	
load_from_disk(analytics_file)	
store_to_disk(analytics_folder)	
to_api()	

classmethod from_dict(*data: dict[str, Any]*) → T

For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case. The content attribute does not change (even if

we put it inside client class).

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogDeclarat

class gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.**Catalo**

Bases: *gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogAnalyticsBase*

__init__(*id: str*, *title: str*, *content: dict[str, Any]*, *description: str = None*, *tags: list[str] = None*)

Methods

__init__(id, title, content[, description, tags])	
from_api(entity)	
from_dict(data)	For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case.
get_kwargs()	
load_from_disk(analytics_file)	
store_to_disk(analytics_folder)	
to_api()	

classmethod from_dict(*data: dict[str, Any]*) → T

For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case. The content attribute does not change (even if we put it inside client class).

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogDeclarat

class gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.**Catalog**

Bases: object

__init__(*analytics: Optional[*gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogDecla = None*)

Methods

__init__([analytics])
from_api(entity)
from_dict(data[, camel_case]) **param data** Data loaded for example from the file.
load_from_disk(workspace_folder)
store_to_disk(workspace_folder)
to_api()

classmethod from_dict(*data: dict[str, Any]*, *camel_case: bool = True*) → *CatalogDeclarativeAnalytics*

Parameters

- **data** – Data loaded for example from the file.
- **camel_case** – True if the variable names in the input data are serialized names as specified in the OpenAPI document. False if the variables names in the input data are python variable names in PEP-8 snake case.

Returns CatalogDeclarativeAnalytics object.

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogDeclarat

class gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.**Catalo**

Bases: `object`

__init__(*analytical_dashboards: list[CatalogDeclarativeAnalyticalDashboard] = None*, *dashboard_plugins: list[CatalogDeclarativeDashboardPlugin] = None*, *filter_contexts: list[CatalogDeclarativeFilterContext] = None*, *metrics: list[CatalogDeclarativeMetric] = None*, *visualization_objects: list[CatalogDeclarativeVisualizationObject] = None*)

Methods

__init__([analytical_dashboards, ...])	
from_api(entity)	
get_analytical_dashboards_folder(...)	
get_analytics_model_folder(workspace_folder)	
get_dashboard_plugins_folder(...)	
get_filter_contexts_folder(...)	

continues on next page

Table 99 – continued from previous page

get_metrics_folder(analytics_model_folder)	
get_visualization_objects_folder(...)	
load_from_disk(workspace_folder)	
store_to_disk(workspace_folder)	
to_api()	

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogDeclarat

class gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.**Catalo**

Bases: *gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogAnalyticsBase*

__init__(*id: str*, *title: str*, *content: dict[str, Any]*, *description: str = None*, *tags: list[str] = None*)

Methods

__init__(id, title, content[, description, tags])	
from_api(entity)	
from_dict(data)	For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case.
get_kwargs()	

continues on next page

Table 100 – continued from previous page

load_from_disk(analytics_file)	
store_to_disk(analytics_folder)	
to_api()	

classmethod from_dict(*data: dict[str, Any]*) → T

For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case. The content attribute does not change (even if we put it inside client class).

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogDeclarat

class gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.**Catalo**

Bases: *gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogAnalyticsBase*

__init__(*id: str*, *title: str*, *content: dict[str, Any]*, *description: str = None*, *tags: list[str] = None*)

Methods

__init__(id, title, content[, description, tags])	
from_api(entity)	
from_dict(data)	For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case.
get_kwargs()	

continues on next page

Table 101 – continued from previous page

load_from_disk(analytics_file)	
store_to_disk(analytics_folder)	
to_api()	

classmethod from_dict(*data: dict[str, Any]*) → T

For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case. The content attribute does not change (even if we put it inside client class).

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogDeclarat

class gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.**Catalo**

Bases: *gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogAnalyticsBase*

__init__(*id: str*, *title: str*, *content: dict[str, Any]*, *description: str = None*, *tags: list[str] = None*)

Methods

__init__(id, title, content[, description, tags])	
from_api(entity)	
from_dict(data)	For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case.
get_kwargs()	

continues on next page

Table 102 – continued from previous page

load_from_disk(analytics_file)	
store_to_disk(analytics_folder)	
to_api()	

classmethod from_dict(*data: dict[str, Any]*) → T

For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case. The content attribute does not change (even if we put it inside client class).

gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogDeclarat

class gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.**Catalo**

Bases: *gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogAnalyticsBase*

__init__(*id: str*, *title: str*, *content: dict[str, Any]*, *description: str = None*, *tags: list[str] = None*)

Methods

__init__(id, title, content[, description, tags])	
from_api(entity)	
from_dict(data)	For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case.
get_kwargs()	

continues on next page

Table 103 – continued from previous page

load_from_disk(analytics_file)	
store_to_disk(analytics_folder)	
to_api()	

classmethod from_dict(*data: dict[str, Any]*) → T

For simplification, we can use directly from_api method, because all attributes follow the same attributes name convention, which is same for snake and camel case. The content attribute does not change (even if we put it inside client class).

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model

Modules

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset	
gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.date_dataset	
gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.ldm	

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset

Modules

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset	

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset

Classes

CatalogDataSourceTableIdentifier(id, ...)	
CatalogDeclarativeAttribute(id, title, labels)	
CatalogDeclarativeDataset(id, title, grain, ...)	

continues on next page

Table 106 – continued from previous page

CatalogDeclarativeFact(id, title, source_column)
CatalogDeclarativeLabel(id, title, primary, ...)
CatalogDeclarativeReference(identifier, ...)

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.CatalogDataSource

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.**CatalogDa**

Bases: `object`

__init__(*id: str*, *data_source_id: str*)

Methods

__init__(id, data_source_id)
from_api(entity)
to_api()

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.CatalogDeclarative

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.**CatalogD**

Bases: *gooddata_sdk.catalog.entity.CatalogTitleEntity*

__init__(*id: str*, *title: str*, *labels: list[*CatalogDeclarativeLabel*]*, *description: str = None*, *tags: list[str] = None*)

Methods

`__init__`(id, title, labels[, description, tags])
`from_api`(entity)
`to_api`()

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.CatalogDeclarative

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.**CatalogD**

Bases: *gooddata_sdk.catalog.entity.CatalogTitleEntity*

__init__(*id: str*, *title: str*, *grain: list[CatalogGrainIdentifier]*, *references: list[*CatalogDeclarativeReference*]*, *description: str = None*, *attributes: list[*CatalogDeclarativeAttribute*] = None*, *facts: list[*CatalogDeclarativeFact*] = None*, *data_source_table_id:* CatalogDataSourceTableIdentifier *= None*, *tags: list[str] = None*)

Methods

`__init__`(id, title, grain, references[, ...])	
`from_api`(entity)	
`from_dict`(data[, camel_case])	**param data** Data loaded for example from the file.
`load_from_disk`(dataset_file)	
`store_to_disk`(datasets_folder)	
`to_api`()	

classmethod from_dict(*data: dict[str, Any]*, *camel_case: bool = True*) → *CatalogDeclarativeDataset*

Parameters

- **data** – Data loaded for example from the file.
- **camel_case** – True if the variable names in the input data are serialized names as specified in the OpenAPI document. False if the variables names in the input data are python variable names in PEP-8 snake case.

Returns CatalogDeclarativeDataset object.

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.CatalogDeclarative

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.**CatalogD**

Bases: *gooddata_sdk.catalog.entity.CatalogTitleEntity*

__init__(*id: str*, *title: str*, *source_column: str*, *description: str = None*, *tags: list[str] = None*)

Methods

__init__(id, title, source_column[, ...])
from_api(entity)
to_api()

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.CatalogDeclarative

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.**CatalogD**

Bases: *gooddata_sdk.catalog.entity.CatalogTitleEntity*

__init__(*id: str*, *title: str*, *primary: bool*, *source_column: str*, *description: str = None*, *tags: list[str] = None*, *value_type: str = None*)

Methods

__init__(id, title, primary, source_column)
from_api(entity)
to_api()

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.CatalogDeclarative

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.dataset.dataset.**CatalogD**

Bases: object

__init__(*identifier: CatalogReferenceIdentifier*, *multi_value: bool*, *source_columns: list[str]*)

Methods

__init__(identifier, multi_value, source_columns)
from_api(entity)
to_api()

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.date_dataset

Modules

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.date_dataset.date_dataset

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.date_dataset.date_dataset

Classes

CatalogDeclarativeDateDataset(id, title, ...)
CatalogGranularitiesFormatting(title_base, ...)

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.date_dataset.date_dataset.CatalogI

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.date_dataset.date_dataset

Bases: *gooddata_sdk.catalog.entity.CatalogTitleEntity*

__init__(*id: str*, *title: str*, *granularities_formatting:* CatalogGranularitiesFormatting, *granularities: list[str]*, *description: str = None*, *tags: list[str] = None*)

Methods

__init__(id, title, ...[, description, tags])	
from_api(entity)	
from_dict(data[, camel_case])	**param data** Data loaded for example from the file.
load_from_disk(date_instance_file)	
store_to_disk(date_instances_folder)	
to_api()	

classmethod from_dict(*data: dict[str, Any]*, *camel_case: bool = True*) → *CatalogDeclarativeDateDataset*

Parameters

- **data** – Data loaded for example from the file.
- **camel_case** – True if the variable names in the input data are serialized names as specified in the OpenAPI document. False if the variables names in the input data are python variable names in PEP-8 snake case.

Returns CatalogDeclarativeDateDataset object.

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.date_dataset.date_dataset.Catalog

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.date_dataset.date_dataset

Bases: object

__init__(*title_base: str*, *title_pattern: str*)

Methods

__init__(title_base, title_pattern)	
from_api(entity)	
to_api()	

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.ldm

Classes

CatalogDeclarativeLdm([datasets, date_instances])	
CatalogDeclarativeModel([ldm])	

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.ldm.CatalogDeclarativeLdm

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.ldm.**CatalogDeclarativeLdm**

Bases: `object`

__init__(*datasets: list[CatalogDeclarativeDataset] = None*, *date_instances: list[CatalogDeclarativeDateDataset] = None*)

Methods

__init__([datasets, date_instances])	
from_api(entity)	
get_datasets_folder(ldm_folder)	
get_date_instances_folder(ldm_folder)	
get_ldm_folder(workspace_folder)	
load_from_disk(workspace_folder)	
store_to_disk(workspace_folder)	
to_api()	

gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.ldm.CatalogDeclarativeModel

class gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.ldm.**CatalogDeclarativeMo**

Bases: object

__init__(*ldm: Optional[*gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.ldm.CatalogDeclarativeLdm*] = None*)

Methods

__init__([ldm])	
from_api(entity)	
from_dict(data[, camel_case])	**param data** Data loaded for example from the file.
load_from_disk(workspace_folder)	
modify_mapped_data_source(data_source_mapping)	
store_to_disk(workspace_folder)	
to_api()	

classmethod from_dict(*data: dict[str, Any]*, *camel_case: bool = True*) → *CatalogDeclarativeModel*

Parameters

- **data** – Data loaded for example from the file.
- **camel_case** – True if the variable names in the input data are serialized names as specified in the OpenAPI document. False if the variables names in the input data are python variable names in PEP-8 snake case.

Returns CatalogDeclarativeModel object.

gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace

Classes

CatalogDeclarativeWorkspace(id, name[, ...])	
CatalogDeclarativeWorkspaceDataFilter(id, ...)	
CatalogDeclarativeWorkspaceDataFilterSetting(id, ...)	
CatalogDeclarativeWorkspaceModel([ldm, ...])	
CatalogDeclarativeWorkspaces(workspaces, ...)	

gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.CatalogDeclarativeWorkspace

class gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.**CatalogDeclarativeWorkspace**(

Bases: *gooddata_sdk.catalog.entity.CatalogNameEntity*

__init__(*id: str*, *name: str*, *compute_client: str = None*, *model:* CatalogDeclarativeWorkspaceModel = *None*, *parent: CatalogWorkspaceIdentifier = None*, *permissions: list[CatalogDeclarativeSingleWorkspacePermission] = None*, *hierarchy_permissions: list[CatalogDeclarativeWorkspaceHierarchyPermission] = None*)

Methods

__init__(id, name[, compute_client, model, ...])	
from_api(entity)	
from_dict(data[, camel_case])	**param data** Data loaded for example from the file.
load_from_disk(workspaces_folder, workspace_id)	
store_to_disk(workspaces_folder)	
to_api([include_nested_structures])	

classmethod from_dict(*data: dict[str, Any]*, *camel_case: bool = True*) → *CatalogDeclarativeWorkspace*

Parameters

- **data** – Data loaded for example from the file.
- **camel_case** – True if the variable names in the input data are serialized names as specified in the OpenAPI document. False if the variables names in the input data are python variable names in PEP-8 snake case.

Returns CatalogDeclarativeWorkspace object.

gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.CatalogDeclarativeWorkspaceDataFilt

class gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.**CatalogDeclarativeWorkspaceDa**

Bases: object

__init__(*id: str*, *title: str*, *column_name: str*, *workspace_data_filter_settings: list[*CatalogDeclarativeWorkspaceDataFilterSetting*]*, *description: str = None*, *workspace: CatalogWorkspaceIdentifier = None*)

Methods

__init__(id, title, column_name, ...[, ...])
from_api(entity)
from_dict(data[, camel_case]) **param data** Data loaded for example from the file.
load_from_disk(workspaces_data_filter_file)
store_to_disk(workspaces_data_filters_folder)
to_api()

classmethod from_dict(*data: dict[str, Any]*, *camel_case: bool = True*) → *CatalogDeclarativeWorkspaceDataFilter*

Parameters

- **data** – Data loaded for example from the file.
- **camel_case** – True if the variable names in the input data are serialized names as specified in the OpenAPI document. False if the variables names in the input data are python variable names in PEP-8 snake case.

Returns CatalogDeclarativeWorkspaceDataFilter object.

gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.CatalogDeclarativeWorkspaceDataFilt

class gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.**CatalogDeclarativeWorkspaceDa**

Bases: *gooddata_sdk.catalog.entity.CatalogTitleEntity*

__init__(*id: str*, *title: str*, *filter_values: list[str]*, *workspace: CatalogWorkspaceIdentifier*, *description: str = None*)

Methods

__init__(id, title, filter_values, workspace)
from_api(entity)
to_api()

gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.CatalogDeclarativeWorkspaceModel

class gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.**CatalogDeclarativeWorkspaceM**

Bases: object

__init__(*ldm: Optional[*gooddata_sdk.catalog.workspace.declarative_model.workspace.logical_model.ldm.CatalogDeclarativeLdm*] = None*, *analytics: Optional[*gooddata_sdk.catalog.workspace.declarative_model.workspace.analytics_model.analytics_model.CatalogDecla *= None*)

Methods

__init__([ldm, analytics])
from_api(entity)
load_from_disk(workspace_folder)
store_to_disk(workspace_folder)
to_api()

gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.CatalogDeclarativeWorkspaces

class gooddata_sdk.catalog.workspace.declarative_model.workspace.workspace.**CatalogDeclarativeWorkspaces**

Bases: object

__init__(*workspaces: list[*CatalogDeclarativeWorkspace*]*, *workspace_data_filters: list[*CatalogDeclarativeWorkspaceDataFilter*]*)

Methods

__init__(workspaces, workspace_data_filters)	
from_api(entity)	
from_dict(data[, camel_case])	**param data** Data loaded for example from the file.
load_from_disk(layout_organization_folder)	
store_to_disk(layout_organization_folder)	
to_api()	
workspace_data_filters_folder(...)	
workspaces_folder(layout_organization_folder)	

classmethod from_dict(*data: dict[str, Any]*, *camel_case: bool = True*) → *CatalogDeclarativeWorkspaces*

Parameters

- **data** – Data loaded for example from the file.
- **camel_case** – True if the variable names in the input data are serialized names as specified in the OpenAPI document. False if the variables names in the input data are python variable names in PEP-8 snake case.

Returns CatalogDeclarativeWorkspaces object.

gooddata_sdk.catalog.workspace.entity_model

Modules

gooddata_sdk.catalog.workspace.entity_model.content_objects
gooddata_sdk.catalog.workspace.entity_model.workspace

gooddata_sdk.catalog.workspace.entity_model.content_objects

Modules

gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset
gooddata_sdk.catalog.workspace.entity_model.content_objects.metric

gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset

Classes

CatalogAttribute(entity, labels)
CatalogDataset(entity, attributes, facts)
CatalogFact(entity)
CatalogLabel(entity)

gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogAttribute

class gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.**CatalogAttribute**(*entity: dict[str, Any]*, *labels: list[*CatalogLabe

Bases: *gooddata_sdk.catalog.entity.CatalogEntity*

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

Methods

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

Attributes

dataset
description
granularity
id
labels
obj_id
title
type

gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogDataset

class gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.**CatalogDataset**(*entity: dict[str, Any]*, *attributes: list[*CatalogAttribu*facts: list[*CatalogFact*]*)

Bases: *gooddata_sdk.catalog.entity.CatalogEntity*

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

Methods

__init__(entity, 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.workspace.entity_model.content_objects.dataset.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.workspace.entity_model.content_objects.dataset.CatalogFact

class gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.**CatalogFact**(*entity: dict[str, Any]*)

Bases: *gooddata_sdk.catalog.entity.CatalogEntity*

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

Methods

__init__(entity)
as_computable()

Attributes

description
id
obj_id
title
type

gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogLabel

class gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.**CatalogLabel**(*entity: dict[str, Any]*)

Bases: *gooddata_sdk.catalog.entity.CatalogEntity*

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

Methods

__init__(entity)
as_computable()

Attributes

description
id
obj_id
primary
title
type

gooddata_sdk.catalog.workspace.entity_model.content_objects.metric

Classes

CatalogMetric(entity)

gooddata_sdk.catalog.workspace.entity_model.content_objects.metric.CatalogMetric

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

Bases: *gooddata_sdk.catalog.entity.CatalogEntity*

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

Methods

__init__(entity)
as_computable()

Attributes

description
format
id
obj_id
title
type

gooddata_sdk.catalog.workspace.entity_model.workspace

Classes

CatalogWorkspace(workspace_id, name[, parent_id])	

gooddata_sdk.catalog.workspace.entity_model.workspace.CatalogWorkspace

class gooddata_sdk.catalog.workspace.entity_model.workspace.**CatalogWorkspace**(*workspace_id: str*, *name: str*, *parent_id: Optional[str] = None*)

Bases: *gooddata_sdk.catalog.entity.CatalogNameEntity*

__init__(*workspace_id: str*, *name: str*, *parent_id: Optional[str] = None*)

Methods

__init__(workspace_id, name[, parent_id])	
from_api(entity)	
to_api()	

gooddata_sdk.catalog.workspace.model_container

Classes

CatalogWorkspaceContent(valid_obj_fun, ...)	

gooddata_sdk.catalog.workspace.model_container.CatalogWorkspaceContent

class gooddata_sdk.catalog.workspace.model_container.**CatalogWorkspaceContent**(*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.
create_workspace_content_catalog(...)	
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.Attribute, gooddata_sdk.compute.model.metric.Metric, gooddata_sdk.compute.model.base.Filter, gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogLabel, gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogFact, gooddata_sdk.catalog.workspace.entity_model.content_objects.metric.CatalogMetric, *List[Union[*gooddata_sdk.compute.model.attribute.Attribute, gooddata_sdk.compute.model.metric.Metric, gooddata_sdk.compute.model.base.Filter, gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogLabel, gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogFact, gooddata_sdk.catalog.workspace.entity_model.content_objects.metric.CatalogMetric*]*, gooddata_sdk.compute.model.execution.ExecutionDefinition*]*) → *gooddata_sdk.catalog.workspace.model_container.CatalogWorkspaceContent*

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.base.ObjId*, Dict[str, Dict[str, str]], Dict[str, str]]*) → Optional[*gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogAttribute*]

Get attribute by label id.

get_dataset(*dataset_id: Union[str,* gooddata_sdk.compute.model.base.ObjId*]*) → Optional[*gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.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.base.ObjId*]*) → Optional[*gooddata_sdk.catalog.workspace.entity_model.content_objects.metric.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.workspace.service

Classes

CatalogWorkspaceContentService(api_client)
CatalogWorkspaceService(api_client)

gooddata_sdk.catalog.workspace.service.CatalogWorkspaceContentService

class gooddata_sdk.catalog.workspace.service.**CatalogWorkspaceContentService**(*api_client:* gooddata_sdk.client.GoodDataApiClient

Bases: *gooddata_sdk.catalog.catalog_service_base.CatalogServiceBase*

__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_attributes_catalog(workspace_id)	
get_declarative_analytics_model(workspace_id)	
get_declarative_ldm(workspace_id)	
get_facts_catalog(workspace_id)	
get_full_catalog(workspace_id)	Retrieves catalog for a workspace.
get_labels_catalog(workspace_id)	
get_metrics_catalog(workspace_id)	
get_organization()	
layout_organization_folder(layout_root_path)	
layout_workspace_folder(workspace_id, ...)	
load_and_put_declarative_analytics_model(...)	
load_and_put_declarative_ldm(workspace_id[, ...])	
load_declarative_analytics_model(workspace_id)	
load_declarative_ldm(workspace_id[, ...])	
put_declarative_analytics_model(...)	
put_declarative_ldm(workspace_id, ldm[, ...])	
store_declarative_analytics_model(workspace_id)	
store_declarative_ldm(workspace_id[, ...])	

Attributes

`organization_id`

compute_valid_objects(*workspace_id: str*, *ctx: Union[*gooddata_sdk.compute.model.attribute.Attribute, gooddata_sdk.compute.model.metric.Metric, gooddata_sdk.compute.model.base.Filter, gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogLabel, gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogFact, gooddata_sdk.catalog.workspace.entity_model.content_objects.metric.CatalogMetric, *List[Union[*gooddata_sdk.compute.model.attribute.Attribute, gooddata_sdk.compute.model.metric.Metric, gooddata_sdk.compute.model.base.Filter, gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogLabel, gooddata_sdk.catalog.workspace.entity_model.content_objects.dataset.CatalogFact, gooddata_sdk.catalog.workspace.entity_model.content_objects.metric.CatalogMetric*]]*, gooddata_sdk.compute.model.execution.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.workspace.model_container.CatalogWorkspaceContent*

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

Parameters **workspace_id** – workspace identifier

Returns

gooddata_sdk.catalog.workspace.service.CatalogWorkspaceService

class gooddata_sdk.catalog.workspace.service.**CatalogWorkspaceService**(*api_client:* gooddata_sdk.client.GoodDataApiClient)

Bases: *gooddata_sdk.catalog.catalog_service_base.CatalogServiceBase*

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

Methods

__init__(api_client)	
create_or_update(workspace)	
delete_workspace(workspace_id)	This method is implemented according to our implementation of delete workspace, which returns HTTP 204 no matter if the workspace_id exists.
get_declarative_workspace(workspace_id)	
get_declarative_workspaces()	
get_organization()	
get_workspace(workspace_id)	Gets workspace content and returns it as CatalogWorkspace object.
layout_organization_folder(layout_root_path)	
list_workspaces()	
load_and_put_declarative_workspaces([...])	
load_declarative_workspaces([layout_root_path])	
put_declarative_workspace(workspace_id, ...)	
put_declarative_workspaces(workspace)	
store_declarative_workspaces([layout_root_path])	

Attributes

organization_id

delete_workspace(*workspace_id: str*) → None

This method is implemented according to our implementation of delete workspace, which returns HTTP 204 no matter if the workspace_id exists.

get_workspace(*workspace_id: str*) → *gooddata_sdk.catalog.workspace.entity_model.workspace.CatalogWorkspace*

Gets workspace content and returns it as CatalogWorkspace object. :param workspace_id: An input string parameter of workspace id. :return: CatalogWorkspace object containing structure of workspace.

3.1.2 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

3.1.3 gooddata_sdk.compute

Modules

gooddata_sdk.compute.model

gooddata_sdk.compute.service

gooddata_sdk.compute.model

Modules

gooddata_sdk.compute.model.attribute

gooddata_sdk.compute.model.base

gooddata_sdk.compute.model.execution

gooddata_sdk.compute.model.filter

gooddata_sdk.compute.model.metric

gooddata_sdk.compute.model.attribute

Classes

Attribute(local_id, label)

gooddata_sdk.compute.model.attribute.Attribute

class gooddata_sdk.compute.model.attribute.**Attribute**(*local_id: str*, *label: Union[*gooddata_sdk.compute.model.base.ObjId, *str]*)

Bases: *gooddata_sdk.compute.model.base.ExecModelEntity*

__init__(*local_id: str*, *label: Union[*gooddata_sdk.compute.model.base.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.base

Classes

ExecModelEntity()
Filter()
ObjId(id, type)

gooddata_sdk.compute.model.base.ExecModelEntity

class gooddata_sdk.compute.model.base.**ExecModelEntity**

Bases: object

__init__() → None

Methods

__init__()
as_api_model()

gooddata_sdk.compute.model.base.Filter

class gooddata_sdk.compute.model.base.**Filter**
Bases: *gooddata_sdk.compute.model.base.ExecModelEntity*

__init__() → None

Methods

__init__()
as_api_model()
is_noop()

Attributes

apply_on_result

gooddata_sdk.compute.model.base.ObjId

class gooddata_sdk.compute.model.base.**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.execution

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.execution.compute_model_to_api_model

gooddata_sdk.compute.model.execution.**compute_model_to_api_model**(*attributes: Optional[list[Attribute]] = None*, *metrics: Optional[list[Metric]] = None*, *filters: Optional[list[Filter]] = None*) → 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

ExecutionDefinition(attributes, metrics, ...)	
ExecutionResponse(actions_api, workspace_id, ...)	
ExecutionResult(result)	

gooddata_sdk.compute.model.execution.ExecutionDefinition

class gooddata_sdk.compute.model.execution.**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.model.execution.ExecutionResponse

class gooddata_sdk.compute.model.execution.**ExecutionResponse**(*actions_api: gooddata_afm_client.api.actions_api.ActionsApi*, *workspace_id: str*, *exec_def:* gooddata_sdk.compute.model.execution.ExecutionDefinitio *response: gooddata_afm_client.model.afm_execution_response.AfmEx*

Bases: object

__init__(*actions_api: gooddata_afm_client.api.actions_api.ActionsApi*, *workspace_id: str*, *exec_def:* gooddata_sdk.compute.model.execution.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.model.execution.ExecutionResult

class gooddata_sdk.compute.model.execution.**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

continues on next page

Table 171 – continued from previous page

paging	
paging_count	
paging_offset	
paging_total	

gooddata_sdk.compute.model.filter

Classes

AbsoluteDateFilter(dataset, from_date, to_date)	
AllTimeFilter()	Filter that is semantically equivalent to absent filter.
AttributeFilter(label[, values])	
MetricValueFilter(metric, operator, values)	
NegativeAttributeFilter(label[, values])	
PositiveAttributeFilter(label[, values])	
RankingFilter(metrics, operator, value, ...)	
RelativeDateFilter(dataset, granularity, ...)	

gooddata_sdk.compute.model.filter.AbsoluteDateFilter

class gooddata_sdk.compute.model.filter.**AbsoluteDateFilter**(*dataset:* gooddata_sdk.compute.model.base.ObjId, *from_date: str*, *to_date: str*)

Bases: *gooddata_sdk.compute.model.base.Filter*

__init__(*dataset:* gooddata_sdk.compute.model.base.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.filter.AllTimeFilter

class gooddata_sdk.compute.model.filter.**AllTimeFilter**

Bases: *gooddata_sdk.compute.model.base.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.filter.AttributeFilter

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

Bases: *gooddata_sdk.compute.model.base.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.filter.MetricValueFilter

class gooddata_sdk.compute.model.filter.**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.base.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.filter.NegativeAttributeFilter

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

Bases: *gooddata_sdk.compute.model.filter.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.filter.PositiveAttributeFilter

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

Bases: *gooddata_sdk.compute.model.filter.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.filter.RankingFilter

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

Bases: *gooddata_sdk.compute.model.base.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
operator
value

gooddata_sdk.compute.model.filter.RelativeDateFilter

class gooddata_sdk.compute.model.filter.**RelativeDateFilter**(*dataset:* gooddata_sdk.compute.model.base.ObjId, *granularity: str*, *from_shift: int*, *to_shift: int*)

Bases: *gooddata_sdk.compute.model.base.Filter*

__init__(*dataset:* gooddata_sdk.compute.model.base.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.metric

Classes

ArithmeticMetric(local_id, operator, operands)	
Metric(local_id)	
PopDate(attribute, periods_ago)	
PopDateDataset(dataset, periods_ago)	
PopDateMetric(local_id, metric, date_attributes)	
PopDatesetMetric(local_id, metric, date_datasets)	
SimpleMetric(local_id, item[, aggregation, ...])	

gooddata_sdk.compute.model.metric.ArithmeticMetric

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

Bases: *gooddata_sdk.compute.model.metric.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.metric.Metric

class gooddata_sdk.compute.model.metric.**Metric**(*local_id: str*)

Bases: *gooddata_sdk.compute.model.base.ExecModelEntity*

__init__(*local_id: str*) → None

Methods

__init__(local_id)
as_api_model()

Attributes

local_id

gooddata_sdk.compute.model.metric.PopDate

class gooddata_sdk.compute.model.metric.**PopDate**(*attribute: Union[gooddata_sdk.compute.model.base.ObjId, gooddata_sdk.compute.model.attribute.Attribute], periods_ago: int*)

Bases: object

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

Methods

__init__(attribute, periods_ago)
as_api_model()

Attributes

attribute
periods_ago

gooddata_sdk.compute.model.metric.PopDateDataset

class gooddata_sdk.compute.model.metric.**PopDateDataset**(*dataset: Union[gooddata_sdk.compute.model.base.ObjId, str]*, *periods_ago: int*)

Bases: `object`

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

Methods

__init__(dataset, periods_ago)
as_api_model()

Attributes

dataset
periods_ago

gooddata_sdk.compute.model.metric.PopDateMetric

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

Bases: *gooddata_sdk.compute.model.metric.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.metric.PopDatesetMetric

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

Bases: *gooddata_sdk.compute.model.metric.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.metric.SimpleMetric

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

Bases: *gooddata_sdk.compute.model.metric.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.compute.service

Classes

ComputeService(api_client)	Compute service drives computation of analytics for a GoodData.CN workspaces.

gooddata_sdk.compute.service.ComputeService

class gooddata_sdk.compute.service.**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.model.execution.ExecutionDefinition) → *gooddata_sdk.compute.model.execution.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

3.1.4 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 215 – 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

3.1.5 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_data_source`
`catalog_organization`
`catalog_workspace`
`catalog_workspace_content`
`compute`
`insights`
`support`
`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.

3.1.6 gooddata_sdk.support

Classes

SupportService(api_client)	

gooddata_sdk.support.SupportService

class gooddata_sdk.support.**SupportService**(*api_client:* gooddata_sdk.client.GoodDataApiClient)

Bases: `object`

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

Methods

__init__(api_client)	
wait_till_available(timeout[, sleep_time])	Wait till GD.CN service is available. When timeout is:

Attributes

is_available	Checks if GD.CN is available.

property is_available: bool

Checks if GD.CN is available. Can raise exceptions in case of authentication or authorization failure. :return: True - available, False - not available

wait_till_available(*timeout: int*, *sleep_time: float = 2.0*) → None

Wait till GD.CN service is available. When timeout is:

- > 0 exception is raised after given number of seconds.
- = 0 exception is raised whe service is not available immediately
- < 0 no timeout

Method propagates is_available exceptions. :param timeout: seconds to wait to service to be available (see method description for details) :param sleep_time: seconds to wait between GD.CN availability tests

3.1.7 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.model.execution.ExecutionResponse, *first_page:* gooddata_sdk.compute.model.execution.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.model.execution.ExecutionResponse, *first_page:* gooddata_sdk.compute.model.execution.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.
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])	

3.1.8 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_converter[, 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_converter: 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: 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_converter[, 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_converter: 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: 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_converter[, 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_converter: 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: 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

3.1.9 gooddata_sdk.utils

Functions

create_directory(path)	
get_sorted_yaml_files(folder)	
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.
read_layout_from_file(path)	
write_layout_to_file(path, content)	

gooddata_sdk.utils.create_directory

gooddata_sdk.utils.**create_directory**(*path: pathlib.Path*) → None

gooddata_sdk.utils.get_sorted_yaml_files

gooddata_sdk.utils.**get_sorted_yaml_files**(*folder: Path*) → list[Path]

gooddata_sdk.utils.id_obj_to_key

gooddata_sdk.utils.**id_obj_to_key**(*id_obj: Union[str, gooddata_sdk.compute.model.base.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

gooddata_sdk.utils.read_layout_from_file

gooddata_sdk.utils.**read_layout_from_file**(*path: pathlib.Path*) → Any

gooddata_sdk.utils.write_layout_to_file

gooddata_sdk.utils.**write_layout_to_file**(*path: Path*, *content: Union[dict[str, Any], list[dict]]*) → None

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)

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

L

M

N

O

P

R

S

T

V

W