Tamr - Python Client Documentation

Release 0.11

Tamr

Apr 05, 2020

CONTENTS

View on Github

CHAPTER

ONE

EXAMPLE

```
from tamr_unify_client import Client
from tamr_unify_client.auth import UsernamePasswordAuth
import os

# grab credentials from environment variables
username = os.environ['TAMR_USERNAME']
password = os.environ['TAMR_PASSWORD']
auth = UsernamePasswordAuth(username, password)

host = 'localhost' # replace with your Tamr host
tamr = Client(auth, host=host)

# programmatically interact with Tamr!
# e.g. refresh your project's Unified Dataset
project = tamr.projects.by_resource_id('3')
ud = project.unified_dataset()
op = ud.refresh()
assert op.succeeded()
```

CHAPTER

TWO

USER GUIDE

2.1 FAQ

2.1.1 What version of the Python Client should I use?

The Python Client just cares about features, and will try everything it knows to implement those features correctly, independent of the API version.

If you are starting a new project or your existing project does not yet use the Python Client, we encourage you to use the **latest stable version** of the Python Client.

Otherwise, check the Change Log to see:

- what new features and bug fixes are available in newer versions
- which breaking changes (if any) will require changes in your code to get those new features and bug fixes

Note: You do not need to reason about the Tamr API version nor the the Tamr app/server version.

2.1.2 How do I call custom endpoints, e.g. endpoints outside the Tamr API?

To call a custom endpoint *within* the Tamr API, use the `client.request()` method, and provide an endpoint described by a path relative to `base_path`.

For example, if `base_path` is `/api/versioned/v1/` (the default), and you want to get `/api/versioned/v1/projects/1`, you only need to provide `projects/1` (the relative ID provided by the project) as the endpoint, and the Client will resolve that into `/api/versioned/v1/projects/1`.

There are various APIs outside the `/api/versioned/v1/` prefix that are often useful or necessary to call - e.g. `/api/service/health`, or other un-versioned / unsupported APIs. To call a custom endpoint *outside* the Tamr API, use the `client.request()` method, and provide an endpoint described by an *absolute* path (a path starting with `/`). For example, to get `/api/service/health` (no matter what `base_path` is), call `client.request()` with `/api/service/health` as the endpoint. The Client will ignore `base_path` and send the request directly against the absolute path provided.

For additional detail, see [Raw HTTP requests and Unversioned API Access](<user-guide/advanced-usage:Raw HTTP requests and Unversioned API Access>)

2.2 Installation

`tamr-unify-client` is compatible with Python 3.6 or newer.

2.2.1 Stable releases

Installation is as simple as:

```
pip install tamr-unify-client
```

Or:

```
poetry add tamr-unify-client
```

Note: If you don't use poetry, we recommend you use a virtual environment for your project and install the Python Client into that virtual environment.

You can create a virtual environment with Python 3 via:

```
python3 -m venv my-venv
```

For more, see The Hitchhiker's Guide to Python.

2.2.2 Latest (unstable)

Note: This project uses the new `pyproject.toml` file, not a `setup.py` file, so make sure you have the latest version of `pip` installed: `` `pip install -U pip ``.

To install the bleeding edge:

```
git clone https://github.com/Datatamer/tamr-client
cd tamr-client
pip install .
```

2.2.3 Offline installs

First, download `tamr-unify-client` and its dependencies on a machine with online access to PyPI:

```
pip download tamr-unify-client -d tamr-unify-client-requirements
zip -r tamr-unify-client-requirements.zip tamr-unify-client-requirements
```

Then, ship the `.zip` file to the target machine where you want `tamr-unify-client` installed. You can do this via email, cloud drives, `scp` or any other mechanism.

Finally, install `tamr-unify-client` from the saved dependencies:

```
unzip tamr-unify-client-requirements.zip
pip install --no-index --find-links=tamr-unify-client-requirements tamr-unify-client
```

If you are not using a virtual environment, you may need to specify the `--user` flag if you get permissions errors:

```
pip install --user --no-index --find-links=tamr-unify-client-requirements tamr-unify-
↪client
```

2.3 Quickstart

2.3.1 Client configuration

Start by importing the Python Client and authentication provider:

```
from tamr_unify_client import Client
from tamr_unify_client.auth import UsernamePasswordAuth
```

Next, create an authentication provider and use that to create an authenticated client:

```
import os

username = os.environ['TAMR_USERNAME']
password = os.environ['TAMR_PASSWORD']

auth = UsernamePasswordAuth(username, password)
tamr = Client(auth)
```

Warning: For security, it's best to read your credentials in from environment variables or secure files instead of hardcoding them directly into your code.

For more, see User Guide > Secure Credentials.

By default, the client tries to find the Tamr instance on `localhost`. To point to a different host, set the host argument when instantiating the Client.

For example, to connect to `10.20.0.1`:

```
tamr = Client(auth, host='10.20.0.1')
```

2.3.2 Top-level collections

The Python Client exposes 2 top-level collections: Projects and Datasets.

You can access these collections through the client and loop over their members with simple `for`-loops.

E.g.:

```
for project in tamr.projects:
    print(project.name)

for dataset in tamr.datasets:
    print(dataset.name)
```

2.3.3 Fetch a specific resource

If you know the identifier for a specific resource, you can ask for it directly via the `by_resource_id` methods exposed by collections.

E.g. To fetch the project with ID `'1'`:

```
project = tamr.projects.by_resource_id('1')
```

Similarly, if you know the name of a specific resource, you can ask for it directly via the `by_name` methods exposed by collections.

E.g. To fetch the project with name `'Number 1'`:

```
project = tamr.projects.by_name('Number 1')
```

Note: If working with projects across Tamr instances for migrations or promotions, use external IDs (via `by_external_id`) instead of name (via `by_name`).

2.3.4 Resource relationships

Related resources (like a project and its unified dataset) can be accessed through specific methods.

E.g. To access the Unified Dataset for a particular project:

```
ud = project.unified_dataset()
```

2.3.5 Kick-off Tamr Operations

Some methods on Model objects can kick-off long-running Tamr operations.

Here, kick-off a "Unified Dataset refresh" operation:

```
operation = project.unified_dataset().refresh()
assert op.succeeded()
```

By default, the API Clients expose a synchronous interface for Tamr operations.

2.4 Secure Credentials

This section discusses ways to pass credentials securely to `UsernamePasswordAuth`. Specifically, you **should not** hardcode your password(s) in your source code. Instead, you should use environment variables or secure files to store your credentials and simple Python code to read your credentials.

2.4.1 Environment variables

You can use `os.environ` to read in your credentials from environment variables:

```
# my_script.py
import os

from tamr_unify_client.auth import UsernamePasswordAuth

username = os.environ['TAMR_USERNAME'] # replace with your username environment␣
↪variable name
password = os.environ['TAMR_PASSWORD'] # replace with your password environment␣
↪variable name
```

(continues on next page)

(continued from previous page)

```
auth = UsernamePasswordAuth(username, password)
```

You can pass in the environment variables from the terminal by including them before your command:

```
TAMR_USERNAME="my Tamr username" TAMR_PASSWORD="my Tamr password" python my_script.py
```

You can also create an `.sh` file to store your environment variables and simply `source` that file before running your script.

2.4.2 Config files

You can also store your credentials in a secure credentials file:

```
# credentials.yaml
---
username: "my tamr username"
password: "my tamr password"
```

Then `pip install pyyaml` read the credentials in your Python code:

```
# my_script.py
from tamr_unify_client.auth import UsernamePasswordAuth
import yaml

with open("path/to/credentials.yaml") as f: # replace with your credentials.yaml path
  creds = yaml.safe_load(f)

auth = UsernamePasswordAuth(creds['username'], creds['password'])
```

As in this example, we recommend you use YAML as your format since YAML has support for comments and is more human-readable than JSON.

Important: You **should not** check these credentials files into your version control system (e.g. `git`). Do not share this file with anyone who should not have access to the password stored in it.

2.5 Workflows

2.5.1 Continuous Categorization

```
from tamr_unify_client import Client
from tamr_unify_client.auth import UsernamePasswordAuth
import os

username = os.environ['TAMR_USERNAME']
password = os.environ['TAMR_PASSWORD']
auth = UsernamePasswordAuth(username, password)

host = 'localhost' # replace with your host
tamr = Client(auth)
```

(continues on next page)

(continued from previous page)

```
project_id = "1" # replace with your project ID
project = tamr.projects.by_resource_id(project_id)
project = project.as_categorization()

unified_dataset = project.unified_dataset()
op = unified_dataset.refresh()
assert op.succeeded()

model = project.model()
op = model.train()
assert op.succeeded()

op = model.predict()
assert op.succeeded()
```

2.5.2 Continuous Mastering

```
from tamr_unify_client import Client
from tamr_unify_client.auth import UsernamePasswordAuth
import os

username = os.environ['TAMR_USERNAME']
password = os.environ['TAMR_PASSWORD']
auth = UsernamePasswordAuth(username, password)

host = 'localhost' # replace with your host
tamr = Client(auth)

project_id = "1" # replace with your project ID
project = tamr.projects.by_resource_id(project_id)
project = project.as_mastering()

unified_dataset = project.unified_dataset()
op = unified_dataset.refresh()
assert op.succeeded()

op = project.pairs().refresh()
assert op.succeeded()

model = project.pair_matching_model()
op = model.train()
assert op.succeeded()

op = model.predict()
assert op.succeeded()

op = project.record_clusters().refresh()
assert op.succeeded()

op = project.published_clusters().refresh()
assert op.succeeded()
```

2.6 Creating and Modifying Resources

2.6.1 Creating resources

Resources, such as projects, dataset, and attribute configurations, can be created through their respective collections. Each `create` function takes in a dictionary that conforms to the Tamr Public Docs for creating that resource type:

```
spec = {
    "name": "project",
    "description": "Mastering Project",
    "type": "DEDUP"
    "unifiedDatasetName": "project_unified_dataset"
}
project = tamr.projects.create(spec)
```

2.6.2 Using specs

These dictionaries can also be created using spec classes.

Each `Resource` has a corresponding `ResourceSpec` which can be used to build an instance of that resource by specifying the value for each property.

The spec can then be converted to a dictionary that can be passed to `create`.

For instance, to create a project:

```
spec = (
    ProjectSpec.new()
    .with_name("Project")
    .with_type("DEDUP")
    .with_description("Mastering Project")
    .with_unified_dataset_name("Project_unified_dataset")
    .with_external_id("tamrProject1")
)
project = tamr.projects.create(spec.to_dict())
```

Calling `with_*` on a spec creates a new spec with the same properties besides the modified one. The original spec is unaltered, so it could be used multiple times:

```
base_spec = (
    ProjectSpec.new()
    .with_type("DEDUP")
    .with_description("Mastering Project")
)

specs = []
for name in project_names:
    spec = (
        base_spec.with_name(name)
        .with_unified_dataset_name(name + "_unified_dataset")
    )
    specs.append(spec)

projects = [tamr.projects.create(spec.to_dict()) for spec in specs]
```

2.6.3 Creating a dataset

Datasets can be created as described above, but the dataset's schema and records must then be handled separately.

To combine all of these steps into one, `DatasetCollection` has a convenience function `create_from_dataframe` that takes a Pandas DataFrame. This makes it easy to create a Tamr dataset from a CSV:

```
import pandas as pd

df = pd.read_csv("my_data.csv", dtype=str)      # string is the recommended data type
dataset = tamr.datasets.create_from_dataframe(df, primary_key_name="primary key name",
↪ dataset_name="My Data")
```

This will create a dataset called "My Data" with the specified primary key, an attribute for each column of the `DataFrame`, and the `DataFrame`'s rows as records.

2.6.4 Modifying a resource

Certain resources can also be modified using specs.

After getting a spec corresponding to a resource and modifying some properties, the updated resource can be committed to Tamr with the `put` function:

```
updated_dataset = (
    dataset.spec()
    .with_description("Modified description")
    .put()
)
```

Each spec class has many properties that can be changed, but refer to the Public Docs for which properties will actually be updated in Tamr. If an immutable property is changed in the update request, the new value will simply be ignored.

2.7 Logging

IMPORTANT Make sure to configure logging BEFORE `import`ing from 3rd party libraries. Logging will use the first configuration it finds, and if a library configures logging before you, your configuration will be ignored.

To configure logging, simply follow the official Python logging HOWTO.

For example:

```
# script.py
import logging

logging.basicConfig(filename="script.log", level=logging.INFO)

# configure logging before other imports

from tamr_unify_client import Client
from tamr_unify_client.auth import UsernamePasswordAuth

auth = UsernamePasswordAuth("my username", "my password")
```

(continues on next page)

(continued from previous page)

```
tamr = Client(auth, host="myhost")

for p in tamr.projects:
    print(p)

for d in tamr.datasets:
    print(d)

# should cause an HTTP error
tamr.get("/invalid/api/path").successful()
```

This will log all API requests made and print the response bodies for any requests with HTTP error codes.

If you want to **only** configure logging for the Tamr Client:

```
import logging
logger = logging.getLogger('tamr_unify_client')
logger.setLevel(logging.INFO)
logger.addHandler(logging.FileHandler('tamr-client.log'))

# configure logging before other imports

from tamr_unify_client import Client
from tamr_unify_client import UsernamePasswordAuth

# rest of script goes here
```

2.8 Geospatial Data

2.8.1 What geospatial data is supported?

In general, the Python Geo Interface is supported; see https://gist.github.com/sgillies/2217756.

There are three layers of information, modeled after GeoJSON (see https://tools.ietf.org/html/rfc7946):

- The outermost layer is a FeatureCollection
- Within a FeatureCollection are Features, each of which represents one "thing", like a building or a river. Each feature has:
  - type (string; required)
  - id (object; required)
  - geometry (Geometry, see below; optional)
  - bbox ("bounding box", 4 doubles; optional)
  - properties (map[string, object]; optional)
- Within a Feature is a Geometry, which represents a shape, like a point or a polygon. Each geometry has:
  - type (one of "Point", "MultiPoint", "LineString", "MultiLineString", "Polygon", "MultiPolygon"; required)
  - coordinates (doubles; exactly how these are structured depends on the type of the geometry)

Although the Python Geo Interface is non-prescriptive when it comes to the data types of the id and properties, Tamr has a more restricted set of supported types. See https://docs.tamr.com/reference#attribute-types.

The `Dataset` class supports the `__geo_interface__` property. This will produce one `FeatureCollection` for the entire dataset.

There is a companion iterator `itergeofeatures()` that returns a generator that allows you to stream the records in the dataset as Geospatial features.

To produce a GeoJSON representation of a dataset:

```
dataset = client.datasets.by_name("my_dataset")
with open("my_dataset.json", "w") as f:
    json.dump(dataset.__geo_interface__, f)
```

By default, `itergeofeatures()` will use the first dataset attribute with geometry type to fill in the feature geometry. You can override this by specifying the geometry attribute to use in the `geo_attr` parameter to `itergeofeatures`.

`Dataset` can also be updated from a feature collection that supports the Python Geo Interface:

```
import geopandas
geodataframe = geopandas.GeoDataFrame(...)
dataset = client.dataset.by_name("my_dataset")
dataset.from_geo_features(geodataframe)
```

By default the features' geometries will be placed into the first dataset attribute with geometry type. You can override this by specifying the geometry attribute to use in the `geo_attr` parameter to `from_geo_features`.

2.8.2 Rules for converting from Tamr records to Geospatial Features

The record's primary key will be used as the feature's `id`. If the primary key is a single attribute, then the value of that attribute will be the value of `id`. If the primary key is composed of multiple attributes, then the value of the `id` will be an array with the values of the key attributes in order.

Tamr allows any number of geometry attributes per record; the Python Geo Interface is limited to one. When converting Tamr records to Python Geo Features, the first geometry attribute in the schema will be used as the geometry; all other geometry attributes will appear as properties with no type conversion. In the future, additional control over the handling of multiple geometries may be provided; the current set of capabilities is intended primarily to support the use case of working with FeatureCollections within Tamr, and FeatureCollection has only one geometry per feature.

An attribute is considered to have geometry type if it has type `RECORD` and contains an attribute named `point`, `multiPoint`, `lineString`, `multiLineString`, `polygon`, or `multiPolygon`.

If an attribute named `bbox` is available, it will be used as `bbox`. No conversion is done on the value of `bbox`. In the future, additional control over the handling of `bbox` attributes may be provided.

All other attributes will be placed in `properties`, with no type conversion. This includes all geometry attributes other than the first.

2.8.3 Rules for converting from Geospatial Features to Tamr records

The Feature's `id` will be converted into the primary key for the record. If the record uses a simple key, no value translation will be done. If the record uses a composite key, then the value of the Feature's `id` must be an array of values, one per attribute in the key.

If the Feature contains keys in `properties` that conflict with the record keys, `bbox`, or geometry, those keys are ignored (omitted).

If the Feature contains a `bbox`, it is copied to the record's `bbox`.

All other keys in the Feature's `properties` are propagated to the same-name attribute on the record, with no type conversion.

2.8.4 Streaming data access

The `Dataset` method `itergeofeatures()` returns a generator that allows you to stream the records in the dataset as Geospatial features:

```
my_dataset = client.datasets.by_name("my_dataset")
for feature in my_dataset.itergeofeatures():
    do_something(feature)
```

Note that many packages that consume the Python Geo Interface will be able to consume this iterator directly. For example::

```
from geopandas import GeoDataFrame
df = GeoDataFrame.from_features(my_dataset.itergeofeatures())
```

This allows construction of a GeoDataFrame directly from the stream of records, without materializing the intermediate dataset.

2.9 Advanced Usage

2.9.1 Asynchronous Operations

You can opt-in to an asynchronous interface via the asynchronous keyword argument for methods that kick-off Tamr operations.

E.g.:

```
op = project.unified_dataset().refresh(asynchronous=True)
# do asynchronous stuff here while operation is running
op = op.wait() # hangs until operation finishes
assert op.succeeded()
```

2.9.2 Raw HTTP requests and Unversioned API Access

We encourage you to use the high-level, object-oriented interface offered by the Python Client. If you aren't sure whether you need to send low-level HTTP requests, you probably don't.

But sometimes it's useful to directly send HTTP requests to Tamr; for example, Tamr has many APIs that are not covered by the higher-level interface (most of which are neither versioned nor supported). You can still call these endpoints using the Python Client, but you'll need to work with raw `Response` objects.

Custom endpoint

The client exposes a `request` method with the same interface as `requests.request`:

```
# import Python Client library and configure your client

tamr = Client(auth)
# do stuff with the `tamr` client

# now I NEED to send a request to a specific endpoint
response = tamr.request('GET', 'relative/path/to/resource')
```

This will send a request relative to the base_path registered with the client. If you provide an absolute path to the resource, the base_path will be ignored when composing the request:

```
# import Python Client library and configure your client

tamr = Client(auth)

# request a resource outside the configured base_path
response = tamr.request('GET', '/absolute/path/to/resource')
```

You can also use the `get`, `post`, `put`, `delete` convenience methods:

```
# e.g. `get` convenience method
response = tamr.get('relative/path/to/resource')
```

Custom Host / Port / Base API path

If you need to repeatedly send requests to another port or base API path (i.e. not `/api/versioned/v1/`), you can simply instantiate a different client.

Then just call `request` as described above:

```
# import Python Client library and configure your client

tamr = api.Client(auth)
# do stuff with the `tamr` client

# now I NEED to send requests to a different host/port/base API path etc..
# NOTE: in this example, we reuse `auth` from the first client, but we could
# have made a new Authentication provider if this client needs it.
custom_client = api.Client(
  auth,
  host="10.10.0.1",
  port=9090,
  base_path="/api/some_service/",
)
response = custom_client.get('relative/path/to/resource')
```

One-off authenticated request

All of the Python Client Authentication providers adhere to the `requests.auth.BaseAuth` interface.

This means that you can pass in an Authentication provider directly to the `requests` library:

```
from tamr_unify_client.auth import UsernamePasswordAuth
import os
import requests
```

(continues on next page)

(continued from previous page)

```
username = os.environ['TAMR_USERNAME']
password =  os.environ['TAMR_PASSWORD']
auth = UsernamePasswordAuth(username, password)

response = requests.request('GET', 'some/specific/endpoint', auth=auth)
```

CHAPTER

THREE

CONTRIBUTOR GUIDE

3.1 Contributor guide

3.1.1 Submitting Bug Reports and Feature Requests

Submit bug reports and feature requests as Github issues .

Check through existing issues (open and closed) to confirm that the bug hasn't been reported before.

If the bug/feature has been submitted already, leave a like on the Github Issue.

3.1.2 Code Migrations

Some of the codebase is old and outdated.

To know which patterns to follow and which to avoid, you can check out ongoing code migrations

3.1.3 Configure your Text Editor

- Atom

3.1.4 Install

This project uses `pyenv` and `poetry`. If you do not have these installed, checkout the toolchain guide.

---

1. Clone your fork and `cd` into the project:

```
git clone https://github.com/<your-github-username>/tamr-client
cd tamr-client
```

2. Set a Python version for this project. Must be Python 3.6+ (e.g. `3.7.3`):

```
pyenv local 3.7.3
```

3. Check that your Python version matches the version specified in `.python-version`:

```
cat .python-version
python --version
```

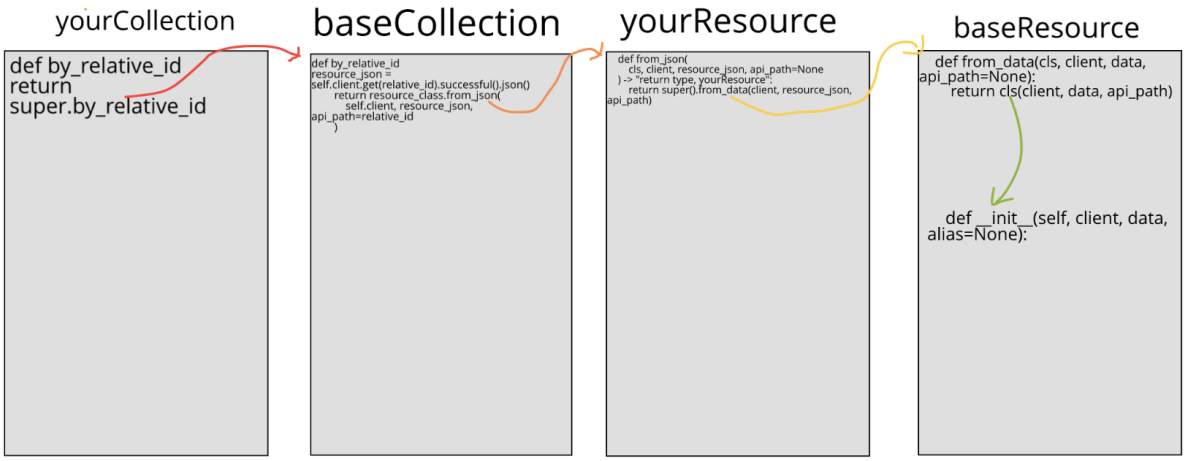
4. Install dependencies via `poetry`:

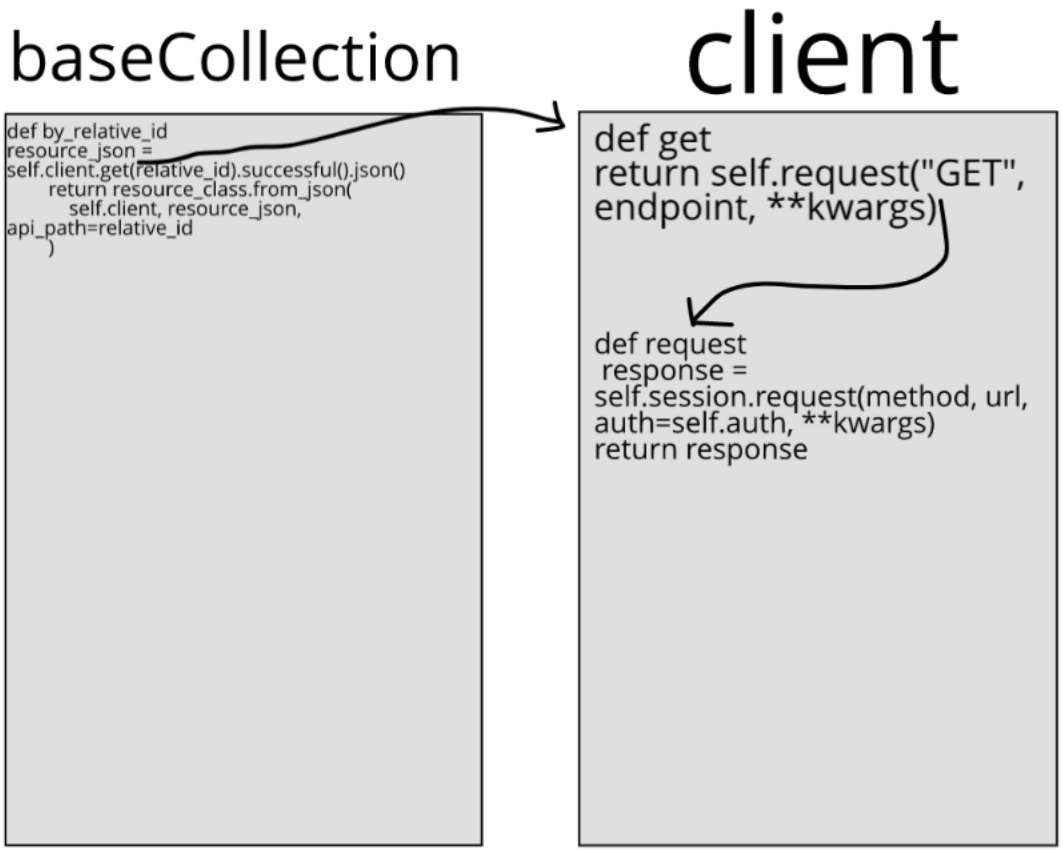
```
poetry install
```

3.1.5 Navigating Inheritance

Older parts of the codebase heavily use inheritance. We are in the process of migrating to `dataclasses` to simplify the codebase, but in the meantime you might want to know how the inheritance machinery we have works.

`yourResource` and `yourCollection` are files that inherit from `baseResource` and `baseCollection`. Examples of such files would be `resource.py` and `collection.py` in the `attribute_configuration` folder under `project`.





Step 1 (red): `yourCollection`'s `by_relative_id` returns `super.by_relative_id`, which comes from `baseCollection`

Step 1a (black): within `by_relative_id`, variable `resource_json` is defined as `self.client.get.[etc]`. `Client`'s `.get` returns `self.request`

Step 1b (black): `client`'s `.request` makes a request to the provided URL (this is the method actually fetching the data)

Step 2 (orange): `baseCollection`'s `by_relative_id` returns `resource_class.from_json`, which is the `from_json` defined in `yourResource`

Step 3 (yellow): `yourResource`'s `from_json` returns `super.from_data`, which comes from `baseResource`

Step 4 (green): `baseResource`'s `from_data` returns `cls` , one of the parameters entered for `from_data`. `cls` is a `yourResource`, because in `from_json` the return type is specified to be a `yourResource`. When `cls` is returned, a `yourResource` that has been filled with the data retrieved in `client`'s `.request` is what comes back.

3.1.6 Pull Requests

For larger, new features:

Open an RFC issue. Discuss the feature with project maintainers to be sure that your change fits with the project vision and that you won't be wasting effort going in the wrong direction.

Once you get the green light from maintainers, you can proceed with the PR.

Contributions / PRs should follow the Forking Workflow:

1. Fork it: `https://github.com/[your-github-username]/tamr-client/fork`
2. Create your feature branch:

```
git checkout -b my-new-feature
```

3. Commit your changes:

```
git commit -am 'Add some feature'
```

4. Push to the branch:

```
git push origin my-new-feature
```

5. Create a new Pull Request

We optimize for PR readability, so please squash commits before and during the PR review process if you think it will help reviewers and onlookers navigate your changes.

Don't be afraid to `push -f` on your PRs when it helps our eyes read your code.

Remember to check for any ongoing code migrations that may be relevant to your PR.

3.1.7 Run and Build

This project uses invoke as its task runner.

Since `invoke` will be running inside of a `poetry` environment, we recommend adding the following alias to your `.bashrc` / `.zshrc` to save you some keystrokes:

```
alias pri='poetry run invoke'
```

Linting & Formatting

To run linter:

```
pri lint # with alias
poetry run invoke lint # without alias
```

To run formatter:

```
pri format # with alias
poetry run invoke format # without alias
```

Run the formatter with the `--fix` flag to autofix formatting:

```
pri format --fix # with alias
poetry run invoke format --fix # without alias
```

Typechecks

To run typechecks:

```
pri typecheck # with alias
poetry run invoke typecheck # without alias
```

Tests

To run all tests:

```
pri test # with alias
poetry run invoke test # without alias
```

To run specific tests, see these pytest docs and run `pytest` explicitly:

```
poetry run pytest tests/unit/test_attribute.py
```

Docs

To build the docs:

```
pri docs # with alias
poetry run invoke docs # without alias
```

After docs are build, view them by:

```
    open -a 'firefox' docs/_build/index.html # open in Firefox
    open -a 'Google Chrome' docs/_build/index.html # open in Chrome
```

3.1.8 Toolchain

This project uses `poetry` as its package manager. For details on `poetry`, see the official documentation.

1. Install pyenv:

   ```
   curl https://pyenv.run | bash
   ```

2. Use `pyenv` to install a compatible Python version (`3.6` or newer; e.g. `3.7.3`):

   ```
   pyenv install 3.7.3
   ```

3. Set that Python version to be your version for this project(e.g. `3.7.3`):

   ```
   pyenv shell 3.7.3
   python --version # check that version matches your specified version
   ```

4. Install `poetry` as described here:

   ```
   curl -sSL https://raw.githubusercontent.com/sdispater/poetry/master/get-poetry.py␣
   ↪| python
   ```

CHAPTER

FOUR

REFERENCE

4.1 Reference

4.1.1 Attributes

Attribute

Attribute Spec

Attribute Collection

Attribute Type

Attribute Type Spec

SubAttribute

4.1.2 Auth

4.1.3 Categorization

Categorization Project

Categories

Category

Category Spec

Category Collection

Taxonomy

4.1.4 Client

4.1.5 Datasets

Dataset

Dataset Spec

Dataset Collection

Dataset Profile

Dataset Status

Dataset URI

Dataset Usage

Dataset Use

4.1.6 Machine Learning Model

4.1.7 Mastering

Binning Model

Estimated Pair Counts

Mastering Project

Published Clusters

Metric

Published Cluster

Published Cluster Configuration

Published Cluster Version

Record Published Cluster

Record Published Cluster Version

4.1.8 Operation

4.1.9 Projects

Attribute Configurations

Attribute Configuration

Attribute Configuration Spec

Attribute Configuration Collection

Attribute Mappings

Attribute Mapping

Attribute Mapping Spec

Attribute Mapping Collection

Project

Project Spec

Project Collection

Project Step

CHAPTER

FIVE

BETA

5.1 BETA

WARNING: Do not rely on BETA features in production workflows. Support from Tamr may be limited.

5.1.1 Reference

Attributes

Attribute

Exceptions

AttributeType

See https://docs.tamr.com/reference#attribute-types

class tamr_client.attribute_type.**Array**(*inner_type*)

Parameters **inner_type** (AttributeType) –

class tamr_client.attribute_type.**Map**(*inner_type*)

Parameters **inner_type** (AttributeType) –

class tamr_client.attribute_type.**Record**(*attributes*)

Parameters **attributes** (Tuple [*SubAttribute*]) –

Type aliases

SubAttribute

class tamr_client.**SubAttribute**(*name*, *type*, *is_nullable*, *description=None*)

Parameters

- **name** (str) –
- **type** (AttributeType) –
- **is_nullable** (bool) –

- **description** (Optional [str]) –

Auth

Datasets

Dataset

Exceptions

Record

Exceptions

Dataframe

Instance

Response

Utilities for working with `requests.Response` .

Session

The `Session` type is an alias for `requests.Session`.

For more information, see the official `requests.Session` docs.

INDEX

T