QuickBase-Client

Tim Kutcher

Aug 09, 2021

CONTENTS:

CHAPTER

ONE

API REFERENCE

1.1 Models

A package for objects in Python and mapping them to QuickBase-expected JSON.

1.1.1 QuickBaseApp

class quickbase_client.**QuickBaseApp**(*app_id: str*, *realm_hostname: str*, *name: str*)

Class for a QuickBase app.

app_id: str

name: str

realm_hostname: str

1.1.2 QuickBaseTable

class quickbase_client.**QuickBaseTable**(***kwargs*)

Base class for a table object.

Variables

- **__dbid__** – The string table ID (the part after /db in the URL).
- **__tablename__** – The english name of the table.
- **__app__** – The *QuickBaseApp* the table belongs to.
- **__reports__** – Lookup of QuickBaseReport objects for this table.

classmethod app_id() → str

Alias to the app's ID.

classmethod client(*user_token: str*)

Factory method to create a *QuickBaseTableClient* for this table.

Parameters **user_token** – The user token for authentication.

classmethod get_attr_from_fid(*fid: int*)

Lookup an attribute name by it's field ID.

Parameters **fid** – The field ID.

classmethod get_field_info(*attr: str*) → quickbase_client.orm.field.QuickBaseField
Get the field info for a given attribute rather than the data.

Parameters **attr** – String name of the attribute.

classmethod get_report(*name: str*)
Get a report by it's name.

Parameters **name** – The name of the report

classmethod realm_hostname() → str
Alias to the app's realm hostname.

1.1.3 QuickBaseField

class quickbase_client.**QuickBaseField**(*fid: int*, *field_type: quickbase_client.orm.field.QuickBaseFieldType*, *label: str = ''*, *formula: Optional[str] = None*)
The metadata for a specific field.

Variables

- **fid** – The field id.
- **field_type** – The *QuickBaseFieldType* of the field.
- **label** – The label for the field.
- **formula** – The QuickBase string formula.

1.1.4 QuickBaseFieldType

class quickbase_client.**QuickBaseFieldType**(*value*)
An Enumeration of Field Types.

Note: Formula fields use the underlying type.

In the generated classes, the import for this class is usually aliased as `Qb` to make it easier to write like `Qb.TEXT`.

These also all have constants in `quickbase_client.orm.field` module prefixed with `QB_`.

ADDRESS = 500

CHECKBOX = 400

DATE = 300

DATETIME = 301

DURATION = 303

EMAIL_ADDRESS = 501

NUMERIC = 200

NUMERIC_CURRENCY = 201

NUMERIC_PERCENT = 202

NUMERIC_RATING = 203

OTHER = 900

RICH_TEXT = 103

TEXT = 100

TEXT_MULTILINE = 101

TEXT_MULTIPLE_CHOICE = 102

TEXT_MULTI_SELECT = 102

TIME_OF_DAY = 302

USER = 600

1.2 Querying

A package for objects in Python and mapping them to QuickBase-expected JSON.

1.2.1 QueryBase

class quickbase_client.**QuickBaseQuery**(*where*, *options=None*, *group_by=None*, *sort_by=None*, *select=None*)

A base object for all of the data for a query.

Variables **where** – The where string, e.g. `"{7.EX.'18'}"`

1.2.2 AST Query Building Methods

This module includes functions which create `QuickBaseQuery` objects.

These can be assembled in an AST-like fashion to build a complex query using higher-level english-readable functions rather than going through the query language (note you can always create a `QuickBaseQuery` and provide the where string to use that).

Example:

```
schema = MyTable.schema
my_query = and_(
    eq_(schema.date_opened, schema.date_created),
    on_or_before_(schema.date_closed, date(2020, 11, 16))
)
print(my_query.where) # ({'9'.EX.'_FID_1'}AND{'10'.OBF.'11-16-2020'})
```

All of the methods (except the two conjunction ones), take a `QuickBaseField` and a value as a parameter. If you pass a *QuickBaseField* for the value, it will compare to the actual field (see above). But note if you pass an attribute of a QuickBaseTable class it would be the value in memory of that attribute. If you want to compare to the actual field, use the schema property of the table or *quickbase_client.QuickBaseTable.get_field_info()*.

Note all of these methods are named with a trailing _ to maintain consistency and never clash with a python keyword or anything.

quickbase_client.query.**or_**(**clauses*)
Conjunction to join 2 or more logical `OR`'s.

quickbase_client.query.**and_**(**clauses*)
Conjunction to join 2 or more logical `AND`'s.

quickbase_client.query.**contains_**(*field*, *val*)
Contains (CT).

quickbase_client.query.**not_contains_**(*field*, *val*)
Not Contains (XCT).

quickbase_client.query.**has_**(*field*, *val*)
Has (HAS).

quickbase_client.query.**not_has_**(*field*, *val*)
Not Has (XHAS).

quickbase_client.query.**eq_**(*field*, *val*)
Equal/Exactly (EX).

quickbase_client.query.**not_eq_**(*field*, *val*)
Not Equal (XEX).

quickbase_client.query.**starts_with_**(*field*, *val*)
Starts With (SW).

quickbase_client.query.**not_starts_width_**(*field*, *val*)
Not Starts With (XSW).

quickbase_client.query.**before_**(*field*, *val*)
Before (BF).

quickbase_client.query.**on_or_before_**(*field*, *val*)
On or Before (OBF).

quickbase_client.query.**after_**(*field*, *val*)
After (AF).

quickbase_client.query.**on_or_after_**(*field*, *val*)
On or After (OAF).

quickbase_client.query.**during_**(*field*, *val*)
During (IR).

quickbase_client.query.**not_during_**(*field*, *val*)
Not During (XIR).

quickbase_client.query.**lt_**(*field*, *val*)
Less than (LT).

quickbase_client.query.**lte_**(*field*, *val*)
Less than or Equal (LTE).

quickbase_client.query.**gt_**(*field*, *val*)
Greater than (GT).

quickbase_client.query.**gte_**(*field*, *val*)
Greater than or Equal (GTE).

1.3 Clients

There are two primary "clients".

There is the high-level `QuickBaseTableClient` which wraps the lower-level `QuickBaseApiClient`.

1.3.1 QuickBaseTableClient

class quickbase_client.**QuickBaseTableClient**(*table: Type[quickbase_client.orm.table.QuickBaseTable]*, *user_token*, *agent='python'*, *normalize_unicode=True*)
Class for making API calls relative to a specific QuickBase table.

This includes making calls for the app in general.

All calls (except *query()*) return a `Response` object for the HTTP response.

Note: Pagination is not handled in any of these methods (yet).

Variables

- **table** – The underlying *QuickBaseTable*
- **api** – The wrapped *QuickBaseApiClient*

__init__(*table: Type[quickbase_client.orm.table.QuickBaseTable]*, *user_token*, *agent='python'*, *normalize_unicode=True*)
Create a client instance.

Parameters

- **table** – The table this client is metaphorically "connected" to.
- **user_token** – The user token to authenticate.
- **agent** – The agent header to send in requests.
- **normalize_unicode** – Whether the JSON Serializer should normalize accented characters.

add_record(*rec*, **args*, ***kwargs*)
Aliased to *add_records()* making rec a list.

add_records(*recs: List[Union[quickbase_client.orm.table.QuickBaseTable, Any]]*, *merge_field_id=None*, *fields_to_return=None*)
Add record.

https://developer.quickbase.com/operation/upsert

Parameters

- **recs** – A list of items that are either the raw record data to post, or the *QuickBaseTable* object/record.
- **merge_field_id** – The list of fields to merge on.

- **fields_to_return** – The list of field ID's to return (default None which means all).

get_app()
Get an app.

https://developer.quickbase.com/operation/getApp

get_field(*field: Union[quickbase_client.orm.field.QuickBaseField, int]*)
Get fields for a table.

https://developer.quickbase.com/operation/getField

Parameters **field** – either the field ID or a *QuickBaseField*

get_fields_for_table()
Get fields for a table.

https://developer.quickbase.com/operation/getFields

get_report(*report*)
Get report.

https://developer.quickbase.com/operation/getRepor

Parameters **report** – Either the report name to lookup, the report id, or a QuickBaseReport object.

get_reports_for_table()
Get reports for a table.

https://developer.quickbase.com/operation/getTableReports

get_table()
Get a table.

https://developer.quickbase.com/operation/getTable

get_tables_for_app()
Get an tables for an app.

https://developer.quickbase.com/operation/getAppTables

query(*query_obj: Optional[quickbase_client.query.query_base.QuickBaseQuery] = None*, *raw=False*, *pager: Optional[quickbase_client.client.pager.ResponsePager] = None*)
Do a query.

https://developer.quickbase.com/operation/runQuery.

See *query* for more.

See *ResponsePager* for handling pagination.

Parameters

- **query_obj** – The *QuickBaseQuery* object to use.
- **raw** – If true, returns a requests.Response, else the data is serialized to a table object.
- **pager** – A *ResponsePager* to handle making paginated requests.

run_report(*report*, *skip=None*, *top=None*)
Run report.

https://developer.quickbase.com/operation/runReport.

Parameters **report** – Either the report name to lookup, the report id, or a QuickBaseReport object.

1.3.2 QuickBaseApiClient

class quickbase_client.**QuickBaseApiClient**(*user_token*, *realm_hostname*, *agent='python'*, *allow_deletes=False*)

The lower-level client to make API requests.

Use *request()* to make an arbitrary request that forwards to make_request()

add_records(*table_id*, *data=None*, *merge_field_id=None*, *fields_to_return=None*)

get_app(*app_id*)

get_field(*field_id*, *table_id*)

get_fields_for_table(*table_id*)

get_report(*report_id*, *table_id*)

get_reports_for_table(*table_id*)

get_table(*app_id*, *table_id*)

get_tables_for_app(*app_id*)

query(*table_id*, *fields_to_select=None*, *where_str=None*, *sort_by=None*, *group_by=None*, *options=None*)

request(**args*, ***kwargs*)

run_report(*report_id*, *table_id*, *skip=None*, *top=None*)

1.3.3 RequestFactory

class quickbase_client.client.request_factory.**QuickBaseRequestFactory**(*user_token*, *realm_hostname*, *agent='python'*, *encoder=None*, *allow_deletes=False*)

make_request(*method*, *endpoint*, *additional_headers=None*, *params=None*, *data=None*)

Make a request (synchronously) and return the Response.

Parameters

- **method** – The (string) HTTP method.
- **endpoint** – The endpoint of the API (starting after "v1/" for example).
- **additional_headers** – A dict of extra headers.
- **params** – Query parameters.
- **data** – The data to send in the body.

1.3.4 ResponsePager

class quickbase_client.**ResponsePager**

Object to pass to methods (query) to manage pagination.

When calling something like QuickBaseTableClient.query, you can pass a ResponsePager, and repeatedly make requests while *more_remaining()* is True.

```
pager = ResponsePager()
while pager.more_remaining():
    recs = my_client.query(pager=pager)
```

more_remaining() → bool

Returns true if there is another request to be made.

CHAPTER

TWO

CLI TOOLS

This package includes a simple command line utility `qbc` for running scripts.

The primary script is *model-generate* which can be used to generate the specific model classes.

The main program itself has one command (run) which can be used to run any scripts that are registered with it (the only core one being model-generate right now).

The `--show-stacktrace` flag will re-raise any errors (internal) that may have occurred.

```
$ qbc -h
usage: qbc [-h] [--show-stacktrace] {run} ...

positional arguments:
  {run}              command help
    run              run a script.

optional arguments:
  -h, --help         show this help message and exit
  --show-stacktrace
```

Running `model-generate` you can pass your user token on the command line or set the `QB_USER_TOKEN` environment variable.

```
$ qbc run model-generate -h
usage: qbc run model-generate [-h] -a [APP_URL] [-d [PKG_DIR]] [-t [USER_TOK]] [-i␣
↪[INCLUDE]]

optional arguments:
  -h, --help            show this help message and exit
  -a [APP_URL], --app-url [APP_URL]
                        the URL of the home page of the app
  -d [PKG_DIR], --pkg-dir [PKG_DIR]
                        the directory to put the package in, defaults to "models"
  -t [USER_TOK], --user-tok [USER_TOK]
                        the user token to authenticate - if not provided will read␣
↪from
                        environment variable QB_USER_TOKEN
  -i [INCLUDE], --include [INCLUDE]
                        ID or name of a table to include - can be specified
                        multiple times; if present, excludes all other tables
```

CHAPTER

THREE

ADDITIONAL TOOLS

3.1 Sending logs to Quickbase with QuickbaseLogHandler

It is often appropriate to send some logs to Quickbase, especially when writing scripts that interface with Quickbase. This package includes a log handler as part of Python standard logging, that will send logs in the background.

You supply it with a table client to the logs table, and the handler will send logs to that table in a non-blocking background thread.

By default, the class assumes the table has attributes for "when", "level", and "message". You can supply a custom record factory via passing a function to *with_record_factory*, or by extending this class and overriding *record_factory*.

For example, say you have a QuickBaseTable class called MyLog. You can create a class like so:

```
 class InvocationLog(QuickBaseTable):
     __dbid__ = 'abcdef'
     __app__ = QuickBaseApp(app_id='aaappp', name='APP', realm_hostname='foo.
↪quickbase.com')

     date_created = QuickBaseField(fid=1, field_type=Qb.DATETIME)
     date_modified = QuickBaseField(fid=2, field_type=Qb.DATETIME)
     recordid = QuickBaseField(fid=3, field_type=Qb.NUMERIC)
     record_owner = QuickBaseField(fid=4, field_type=Qb.USER)
     last_modified = QuickBaseField(fid=5, field_type=Qb.USER)

     log_message = QuickBaseField(fid=6, field_type=Qb.TEXT)
     logged_when = QuickBaseField(fid=7, field_type=Qb.DATETIME)


class MyLogHandler(QuickbaseLogHandler):
    def __init__(self):
        super().__init__(MyLog.client(my_user_token))

    def record_factory(self, record: logging.LogRecord):
        return MyLog(log_message=record.msg, logged_when=datetime.utcnow())


logger = logging.getLogger()
logger.addHandler(MyLogHandler())
logger.info('This message should show up in Quickbase!')
```

class quickbase_client.tools.**QuickbaseLogHandler**(*logs_table_client: quickbase_client.client.table_client.QuickBaseTableClient*)

Class for sending logs to a specific Quickbase table.

You supply it with a table client to the logs table, and the handler will send logs to that table in a non-blocking background thread.

This uses the higher-level QuickbaseTableClient APIs. So you will have to create a class for your table you want to send logs to.

emit(*record*)

Calls *record_factory()* and starts a separate thread to send it to Quickbase.

record_factory(*record: logging.LogRecord*)

Create a QuickBaseTable record object given a LogRecord. By default, this assumes the associated table, under the handlers table client, has properties *when*, *level*, and *message*.

Parameters **record** – The logging.LogRecord.

Returns A QuickBaseTable record object.

CHAPTER

FOUR

QUICK START

4.1 Installation

Installation can be done through pip:

```
pip install quickbase-client
```

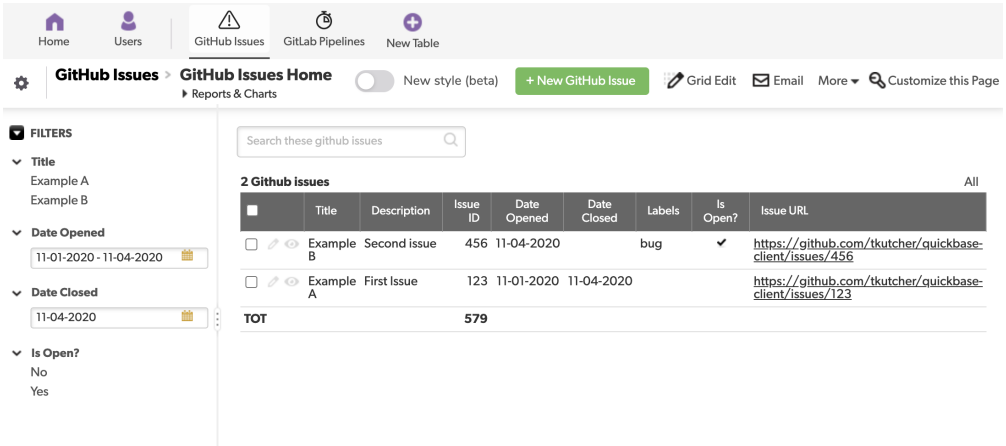
This will install both the library `quickbase_client`, and a command line tool `qbc` for running some handy scripts.

4.1.1 Generating your Models

To interact and authenticate with your QuickBase applications you need a User Token. You can read the QuickBase documentation here on how to create one. It is recommended to set an environment variable `QB_USER_TOKEN` with this value:

```
export QB_USER_TOKEN=mytokenfromquickbase;
```

Next, say you have a hypothetical QuickBase Application named MyApp at `https://foo.quickbase.com/db/abcdef` that has tables for tracking things against a repository like Issues & Pipelines.



Running the following:

```
qbc run model-generate -a https://foo.quickbase.com/db/abcdef
```

Would generate a directory structure like

```
models
├── __init__.py
└── my_app
    ├── __init__.py
    ├── app.py
    ├── github_issue.py
    └── gitlab_pipeline.py
```

And classes like `GitHubIssue` where you can interact with the data model through a Python object.

4.1.2 Writing Records to QuickBase

Classes like `GitHubIssue` that subclass `QuickBaseTable` also get a factory class-method `client(user_tok)` which creates an instance of the higher-level `QuickBaseTableClient` to make API requests for things related to that table:

```
client = GitHubIssue.client(user_tok=os.environ['QB_USER_TOKEN'])
new_issue = GitHubIssue(
    title='Something broke',    # you get friendly-kwargs for fields without worrying␣
↪about ID's
    description='Please fix!',
    date_opened=date.today()    # things like Python date objects will be serialized
)
response = client.add_record(new_issue)
print(response.json())  # all methods (except for query) return the requests Response␣
↪object
```

4.1.3 Querying Records from QuickBase

You can also use the client object to send queries to the QuickBase API through the `query` method. This method will serialize the data back in to a Python object. The *query* method on the table class takes a `QuickBaseQuery` object which is high level wrapper around the parameters needed to make a query.

Notably, the `where` parameter for specifying the query string. There is one (and in the future there will be more) implementation of this which allows you to build query-strings through higher-level python functions.

You can use the methods exposed in the `quickbase_client.query` module like so:

```
# convention to append an underscore to these methods to avoid clashing
# with any python keywords
from quickbase_client.query import on_or_before_
from quickbase_client.query import eq_
from quickbase_client.query import and_

schema = GitHubIssue.schema
q = and_(
    eq_(schema.date_opened, schema.date_created),
    on_or_before_(schema.date_closed, date(2020, 11, 16))
)
print(q.where)  # ({'9'.EX.'_FID_1'}AND{'10'.OBF.'11-16-2020'})
```

(continues on next page)

(continued from previous page)

```
recs = client.query(q)  # recs will be GitHubIssue objects unless passing raw=True
print([str(r) for r in recs])  # ['<GitHubIssue title="Made And Closed Today" id=
↪"10000">']
```

4.1.4 Controlling Lower-Level API Calls

Lastly, say you want to deal with just posting the specific json/data QuickBase is looking for. The `QuickBaseTableClient` object wraps the lower-level `QuickBaseApiClient` object which has methods for just sending the actual data (with an even lower-level utility `QuickBaseRequestFactory` you could also use). These classes manage hanging on to the user token, and the realm hostname, etc. for each request that is made.

For example, note the signature of `query` in `QuickBaseApiClient`:

```
def query(self, table_id, fields_to_select=None, where_str=None,
          sort_by=None, group_by=None, options=None):
```

You can get to this class by going through the table client: `api = client.api`, or from instantiating it directly `api = QuickBaseApiClient(my_user_token, my_realm)`

With this, we could make the exact same request as before:

```
api = QuickBaseApiClient(user_token='my_token', realm_hostname='foo.quickbase.com')
response = api.query(
    table_id='abcdef',
    where_str="({'9'.EX.'_FID_1'}AND{'10'.OBF.'11-16-2020'})")
data = response.json()
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

q

INDEX

Symbols

A

B

C

D

E

G

H

L

M

N

O

Q

R

S

T

U