# Quickbase-Client

**Tim Kutcher**

**Jul 28, 2023**

# 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.
- **schema** – The object to reference field types (rather than field data).

**classmethod app_id**() → str

Alias to the app's ID.

**classmethod client**(*user_token: str*, ***kwargs*)

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

**Parameters**

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

- **kwargs** – Forwarded to *QuickbaseTableClient*.

**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** (*int*) – The field id.
- **field_type** (QuickbaseFieldType) – The *QuickbaseFieldType* of the field.
- **label** (*str*) – The label for the field.
- **formula** (*Optional[str]*) – 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

### 1.2.1 Query Base Class

**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.

---

**Note:** Current alias of `QuickbaseQuery` for backwards compatibility.

---

**Parameters**

- **where** (*Optional[str]*) – The where string, e.g. `"{7.EX.'18'}"`
- **options** (*dict*) – Additional options to pass to the Quickbase runQuery endpoint.
- **group_by** (*list[dict]*) – The groupBy for the Quickbase runQuery endpoint.
- **sort_by** (*list[dict]*) – The sortBy for the Quickbase runQuery endpoint.
- **select** (*list[int]*) – The list of field ID's to return. Note that Quickbase by default (i.e. if this parameter is left as `None`) only returns the "default" fields for the table.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

quickbase_client.query.ast.**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*, *allow_deletes=False*)

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*

**Parameters**

- **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 so that they can be encoded in Quickbase.
- **allow_deletes** (*bool*) – Whether the client should be allowed to perform delete requests. Defaulted to False for now. But note that this is subject to change in 1.0 if there is a different general preference.

**__init__**(*table: Type[quickbase_client.orm.table.QuickbaseTable]*, *user_token*, *agent='python'*, *normalize_unicode=True*, *allow_deletes=False*)
Initialize self. See help(type(self)) for accurate signature.

**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).

**change_record_owner**(*rid*, *new_owner*)
Use the legacy API to change a Record's owner.

See https://help.quickbase.com/api-guide/change_record_owner.html

**Parameters**

- **rid** – The record ID to change the owner of
- **new_owner** – The email address, or user ID, of the user to change to the owner.

**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.

If some fields are coming back as null, a common "gotcha" is that the Quickbase API by default only returns fields listed as "default" in a table. In that case you would have to explicitly specify a `select` in the query, or you can edit the fields in Quickbase to be default fields.

**Parameters**

- **query_obj** – The *QuickbaseQuery* object to use. Note that this object also specifies the `select`, `group_by`, `sort_by`, etc. So to specify those you need to specify them in the provided *QuickbaseQuery*. See its documentation for more details.
- **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.
- **skip** (*int*) – For paging (see Quickbase API)
- **top** (*int*) – For paging (see Quickbase API)

### 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.

**Note:** Current alias of `QuickBaseApiClient` for backwards compatibility - will be removed in version 1.0

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

**Variables** **legacy_api** – The *QuickbaseLegacyApiClient* for making requests to the XML API.

**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 QuickbaseLegacyApiClient

**class** quickbase_client.**QuickbaseLegacyApiClient**(*user_token: str*, *realm_hostname: str*)

Legacy Client which makes requests to the old XML API.

This operates more generically to send any request to the old API. It's primary purpose is to supplement the JSON API for things not yet supported.

It does include some higher-level methods for things that in particular are not supported in the new API's. If there are others you would like added, submit an Issue or a Merge-Request and we can certainly add it!

Rather than get in to having lxml as an optional dependency, it will just deal with XML data as strings for now. If we want more support of the XML API, then it might be worth it to introduce that.

**Parameters**

- **user_token** (*str*) – The user token for authenticating with the API. Note that the other forms of authentication through the XML API are not supported via this library.
- **realm_hostname** (*str*) – The hostname - like `"foo.quickbase.com"`

**change_record_owner**(*table_id*, *rid*, *new_owner*)

The Quickbase API_ChangeRecordOwner action

See https://help.quickbase.com/api-guide/change_record_owner.html

**Parameters**

- **table_id** – The table ID containing the record
- **rid** – The record ID to change the owner of
- **new_owner** – The email address, or user ID, of the user to change to the owner.

**make_request**(*http_method: str*, *quickbase_action: str*, *endpoint: str*, *request_data_xml_str=''*)

Used as a simple Python interface to the old XML API.

**Parameters**

- **http_method** (*str*) – The HTTP method, like `"post"`
- **quickbase_action** (*str*) – The Quickbase Action to send, like `"API_ChangeRecordOwner"`
- **endpoint** (*str*) – The HTTP Endpoint to call, like `/db/abc123`
- **request_data_xml_str** (*str*) – The data to add to the XML string that gets sent, *inside* of the `qdbapi` element/tag. For now this is all done through strings, in the future it may use lxml and have better capabilities for working with those trees.

### 1.3.4 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*) → requests.models.Response

Make a request (synchronously) and return the Response.

**Parameters**

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

### 1.3.5 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.

---

**Note:** NOTE - currently a bunch of duplicate aliases for QuickBase to Quickbase since this was originally released with everything prefixed as QuickBase. But since Quickbase is branding more to "Quickbase", this will eventually be the main naming for version 1.0 in an effort to keep more consistent.

---

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 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.

**static with_record_factory**(*logs_table_client: quickbase_client.client.table_client.QuickbaseTableClient*, *record_factory*)

Create a logger using a function to make records.

**Parameters**

- **logs_table_client** – The QuickbaseTableClient for the logs to go to.
- **record_factory** – A function which takes a *logging.LogRecord* and creates the relevant record (instance of a `QuickbaseTable`).

CHAPTER

FOUR

# QUICK START

## 4.1 Installation

Installation can be done through pip:

```
pip install quickbase-client
```

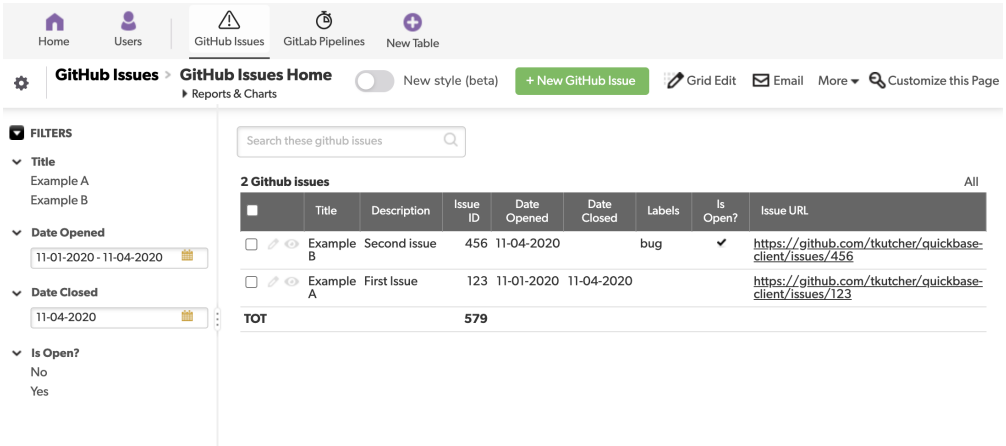
This will install both the library `quickbase_client`, and a command line tool `qbc` for running some handy scripts.

### 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



## W