Queries Documentation

Release 2.1.0

Gavin M. Roy

Aug 07, 2020

Contents

Queries is a BSD licensed opinionated wrapper of the psycopg2 library for interacting with PostgreSQL.

The popular psycopg2 package is a full-featured python client. Unfortunately as a developer, you're often repeating the same steps to get started with your applications that use it. Queries aims to reduce the complexity of psycopg2 while adding additional features to make writing PostgreSQL client applications both fast and easy.

Key features include:

- Simplified API
- Support of Python 2.7+ and 3.4+
- PyPy support via psycopg2cffi
- Asynchronous support for Tornado
- Connection information provided by URI
- Query results delivered as a generator based iterators
- Automatically registered data-type support for UUIDs, Unicode and Unicode Arrays
- Ability to directly access psycopg2 `connection` and `cursor` objects
- Internal connection pooling

CHAPTER 1

Installation

Queries can be installed via the Python Package Index and can be installed by running **`easy_install queries`** or **`pip install queries`**

When installing Queries, `pip` or `easy_install` will automatically install the proper dependencies for your platform.

CHAPTER 2

Contents

2.1 Using Queries

Queries provides both a session based API and a stripped-down simple API for interacting with PostgreSQL. If you're writing applications that will only have one or two queries, the simple API may be useful. Instead of creating a session object when using the simple API methods (`queries.query()` and `queries.callproc()`), this is done for you. Simply pass in your query and the URIs of the PostgreSQL server to connect to:

```
queries.query("SELECT now()", "postgresql://postgres@localhost:5432/postgres")
```

Queries built-in connection pooling will re-use connections when possible, lowering the overhead of connecting and reconnecting. This is also true when you're using Queries sessions in different parts of your application in the same Python interpreter.

2.1.1 Connection URIs

When specifying a URI, if you omit the username and database name to connect with, Queries will use the current OS username for both. You can also omit the URI when connecting to connect to localhost on port 5432 as the current OS user, connecting to a database named for the current user. For example, if your username is *fred* and you omit the URI when issuing `queries.query()` the URI that is constructed would be `postgresql://fred@localhost:5432/fred`.

If you'd rather use individual values for the connection, the queries.uri() method provides a quick and easy way to create a URI to pass into the various methods.

`queries.`**`uri`**(*host='localhost'*, *port=5432*, *dbname='postgres'*, *user='postgres'*, *password=None*)

Return a PostgreSQL connection URI for the specified values.

Parameters

- **`host`** (*str*) – Host to connect to
- **`port`** (*int*) – Port to connect on
- **`dbname`** (*str*) – The database name

- **user** (*str*) – User to connect as
- **password** (*str*) – The password to use, None for no password

Return str The PostgreSQL connection URI

2.1.2 Examples

The following examples demonstrate various aspects of the Queries API. For more detailed examples and documentation, visit the simple, *Session API*, *Query Results*, and *TornadoSession Asynchronous API* pages.

Using queries.uri to generate a URI from individual arguments

```
>>> queries.uri("server-name", 5432, "dbname", "user", "pass")
'postgresql://user:pass@server-name:5432/dbname'
```

Using the queries.Session class

To execute queries or call stored procedures, you start by creating an instance of the *queries.Session* class. It can act as a context manager, meaning you can use it with the `with` keyword and it will take care of cleaning up after itself. For more information on the `with` keyword and context managers, see **PEP 343**.

In addition to both the *queries.Session.query()* and *queries.Session.callproc()* methods that are similar to the simple API methods, the *queries.Session* class provides access to the psycopg2 `connection` and `cursor` objects.

Using queries.Session.query

The following example shows how a *queries.Session* object can be used as a context manager to query the database table:

```
>>> import pprint
>>> import queries
>>>
>>> with queries.Session() as s:
...     for row in s.query('SELECT * FROM names'):
...         pprint.pprint(row)
...
{'id': 1, 'name': u'Jacob'}
{'id': 2, 'name': u'Mason'}
{'id': 3, 'name': u'Ethan'}
```

Using queries.Session.callproc

This example uses *queries.Session.callproc()* to execute a stored procedure and then pretty-prints the single row results as a dictionary:

```
>>> import pprint
>>> import queries
>>> with queries.Session() as session:
...   results = session.callproc('chr', [65])
...   pprint.pprint(results.as_dict())
...
{'chr': u'A'}
```

2.2 Session API

The Session class allows for a unified (and simplified) view of interfacing with a PostgreSQL database server.

Connection details are passed in as a PostgreSQL URI and connections are pooled by default, allowing for reuse of connections across modules in the Python runtime without having to pass around the object handle.

While you can still access the raw psycopg2 `connection` and `cursor` objects to provide ultimate flexibility in how you use the *`queries.Session`* object, there are convenience methods designed to simplify the interaction with PostgreSQL.

For psycopg2 functionality outside of what is exposed in Session, simply use the *`queries.Session.connection`* or *`queries.Session.cursor`* properties to gain access to either object just as you would in a program using psycopg2 directly.

2.2.1 Example Usage

The following example connects to the `postgres` database on `localhost` as the `postgres` user and then queries a table, iterating over the results:

```
import queries

with queries.Session('postgresql://postgres@localhost/postgres') as session:
    for row in session.query('SELECT * FROM table'):
        print row
```

2.2.2 Class Documentation

class queries.**Session**(*uri='postgresql://localhost:5432'*, *cursor_factory=<class 'psycopg2.extras.RealDictCursor'>*, *pool_idle_ttl=60*, *pool_max_size=1*, *autocommit=True*)

The Session class allows for a unified (and simplified) view of interfacing with a PostgreSQL database server. The Session object can act as a context manager, providing automated cleanup and simple, Pythonic way of interacting with the object.

Parameters

- **uri** (*str*) – PostgreSQL connection URI
- **psycopg2.extensions.cursor** – The cursor type to use
- **pool_idle_ttl** (*int*) – How long idle pools keep connections open
- **pool_max_size** (*int*) – The maximum size of the pool to use

backend_pid

Return the backend process ID of the PostgreSQL server that this session is connected to.

Return type int

callproc(*name*, *args=None*)

Call a stored procedure on the server, returning the results in a *`queries.Results`* instance.

Parameters

- **name** (*str*) – The procedure name
- **args** (*list*) – The list of arguments to pass in

Return type *queries.Results*

Raises queries.DataError

Raises queries.DatabaseError

Raises queries.IntegrityError

Raises queries.InternalError

Raises queries.InterfaceError

Raises queries.NotSupportedError

Raises queries.OperationalError

Raises queries.ProgrammingError

close()
Explicitly close the connection and remove it from the connection pool if pooling is enabled. If the connection is already closed

Raises psycopg2.InterfaceError

connection
Return the current open connection to PostgreSQL.

Return type psycopg2.extensions.connection

cursor
Return the current, active cursor for the open connection.

Return type psycopg2.extensions.cursor

encoding
Return the current client encoding value.

Return type str

notices
Return a list of up to the last 50 server notices sent to the client.

Return type list

pid
Return the pool ID used for connection pooling.

Return type str

query(*sql*, *parameters=None*)
A generator to issue a query on the server, mogrifying the parameters against the sql statement. Results are returned as a *queries.Results* object which can act as an iterator and has multiple ways to access the result data.

Parameters

- **sql** (*str*) – The SQL statement
- **parameters** (*dict*) – A dictionary of query parameters

Return type *queries.Results*

Raises queries.DataError

Raises queries.DatabaseError

Raises queries.IntegrityError

Raises queries.InternalError

Raises queries.InterfaceError

Raises queries.NotSupportedError

Raises queries.OperationalError

Raises queries.ProgrammingError

set_encoding(*value='UTF8'*)

Set the client encoding for the session if the value specified is different than the current client encoding.

Parameters **value** (*str*) – The encoding value to use

2.3 Query Results

Results from calls to *Session.query* and *Session.callproc* are returned as an instance of the *Results* class. The *Results* class provides multiple ways to access the information about a query and the data returned from PostgreSQL.

2.3.1 Examples

The following examples illustrate the various behaviors that the *Results* class implements:

Using Results as an Iterator

```
for row in session.query('SELECT * FROM foo'):
    print row
```

Accessing an individual row by index

```
results = session.query('SELECT * FROM foo')
print results[1]  # Access the second row of the results
```

Casting single row results as a dict

```
results = session.query('SELECT * FROM foo LIMIT 1')
print results.as_dict()
```

Checking to see if a query was successful

```
results = session.query("UPDATE foo SET bar='baz' WHERE qux='corgie'")
if results:
    print 'Success'
```

Checking the number of rows by using len(Results)

```
results = session.query('SELECT * FROM foo')
print '%i rows' % len(results)
```

2.3.2 Class Documentation

class queries.**Results**(*cursor*)

The *Results* class contains the results returned from *Session.query* and *Session.callproc*. It is

able to act as an iterator and provides many different methods for accessing the information about and results from a query.

Parameters **cursor** (*psycopg2.extensions.cursor*) – The cursor for the results

as_dict()
Return a single row result as a dictionary. If the results contain multiple rows, a `ValueError` will be raised.

Returns dict

Raises ValueError

count()
Return the number of rows that were returned from the query

Return type int

free()
Used in asynchronous sessions for freeing results and their locked connections.

items()
Return all of the rows that are in the result set.

Return type list

query
Return a read-only value of the query that was submitted to PostgreSQL.

Return type str

rownumber
Return the current offset of the result set

Return type int

status
Return the status message returned by PostgreSQL after the query was executed.

Return type str

2.4 TornadoSession Asynchronous API

Use a Queries Session asynchronously within the Tornado framework.

The `TornadoSession` class is optimized for asynchronous concurrency. Each call to `TornadoSession.callproc` or `TornadoSession.query` grabs a free connection from the connection pool and requires that the results that are r returned as a *Results* object are freed via the *Results.free* method. Doing so will release the free the *Results* object data and release the lock on the connection so that other queries are able to use the connection.

2.4.1 Example Use

The following `RequestHandler` example will return a JSON document containing the query results.

```
import queries
from tornado import gen, web

class ExampleHandler(web.RequestHandler):

```

(continues on next page)

(continued from previous page)

```
    def initialize(self):
        self.session = queries.TornadoSession()

    @gen.coroutine
    def get(self):
        result = yield self.session.query('SELECT * FROM names')
        self.finish({'data': result.items()})
        result.free()
```

See the *Examples* for more `TornadoSession()` examples.

2.4.2 Class Documentation

class queries.tornado_session.**TornadoSession**(*uri='postgresql://localhost:5432'*, *cursor_factory=<class 'psycopg2.extras.RealDictCursor'>*, *pool_idle_ttl=60*, *pool_max_size=25*, *io_loop=None*)

Session class for Tornado asynchronous applications. Uses `tornado.gen.coroutine()` to wrap API methods for use in Tornado.

Utilizes connection pooling to ensure that multiple concurrent asynchronous queries do not block each other. Heavily trafficked services will require a higher `max_pool_size` to allow for greater connection concurrency.

`TornadoSession.query` and `TornadoSession.callproc` must call *`Results.free`*

Parameters

- **uri** (*str*) – PostgreSQL connection URI
- **psycopg2.extensions.cursor** – The cursor type to use
- **pool_idle_ttl** (*int*) – How long idle pools keep connections open
- **pool_max_size** (*int*) – The maximum size of the pool to use

backend_pid

Return the backend process ID of the PostgreSQL server that this session is connected to.

Return type int

callproc(*name*, *args=None*)

Call a stored procedure asynchronously on the server, passing in the arguments to be passed to the stored procedure, yielding the results as a *`Results`* object.

You **must** free the results that are returned by this method to unlock the connection used to perform the query. Failure to do so will cause your Tornado application to run out of connections.

Parameters

- **name** (*str*) – The stored procedure name
- **args** (*list*) – An optional list of procedure arguments

Return type *Results*

Raises queries.DataError

Raises queries.DatabaseError

Raises queries.IntegrityError

Raises queries.InternalError

Raises queries.InterfaceError

Raises queries.NotSupportedError

Raises queries.OperationalError

Raises queries.ProgrammingError

close()
Explicitly close the connection and remove it from the connection pool if pooling is enabled. If the connection is already closed

Raises psycopg2.InterfaceError

connection
Do not use this directly with Tornado applications

Returns

encoding
Return the current client encoding value.

Return type str

notices
Return a list of up to the last 50 server notices sent to the client.

Return type list

pid
Return the pool ID used for connection pooling.

Return type str

query(*sql*, *parameters=None*)
Issue a query asynchronously on the server, mogrifying the parameters against the sql statement and yielding the results as a *Results* object.

You **must** free the results that are returned by this method to unlock the connection used to perform the query. Failure to do so will cause your Tornado application to run out of connections.

Parameters

- **sql** (*str*) – The SQL statement
- **parameters** (*dict*) – A dictionary of query parameters

Return type *Results*

Raises queries.DataError

Raises queries.DatabaseError

Raises queries.IntegrityError

Raises queries.InternalError

Raises queries.InterfaceError

Raises queries.NotSupportedError

Raises queries.OperationalError

Raises queries.ProgrammingError

set_encoding(*value='UTF8'*)
Set the client encoding for the session if the value specified is different than the current client encoding.

Parameters **value** (*str*) – The encoding value to use

validate()
Validate the session can connect or has open connections to PostgreSQL. As of 1.10.3

Deprecated since version 1.10.3: As of 1.10.3, this method only warns about Deprecation

Return type bool

class queries.tornado_session.**Results**(*cursor*, *cleanup*, *fd*)
A TornadoSession specific *queries.Results* class that adds the *Results.free* method. The *Results.free* method **must** be called to free the connection that the results were generated on. *Results* objects that are not freed will cause the connections to remain locked and your application will eventually run out of connections in the pool.

The following examples illustrate the various behaviors that the :queries.Results class implements:

Using Results as an Iterator

```
results = yield session.query('SELECT * FROM foo')
for row in results
    print row
results.free()
```

Accessing an individual row by index

```
results = yield session.query('SELECT * FROM foo')
print results[1]  # Access the second row of the results
results.free()
```

Casting single row results as a dict

```
results = yield session.query('SELECT * FROM foo LIMIT 1')
print results.as_dict()
results.free()
```

Checking to see if a query was successful

```
sql = "UPDATE foo SET bar='baz' WHERE qux='corgie'"
results = yield session.query(sql)
if results:
    print 'Success'
results.free()
```

Checking the number of rows by using len(Results)

```
results = yield session.query('SELECT * FROM foo')
print '%i rows' % len(results)
results.free()
```

as_dict()
Return a single row result as a dictionary. If the results contain multiple rows, a ValueError will be raised.

Returns dict

Raises ValueError

count()
Return the number of rows that were returned from the query

Return type int

free()
Release the results and connection lock from the TornadoSession object. This **must** be called after you finish processing the results from `TornadoSession.query` or `TornadoSession.callproc` or the connection will not be able to be reused by other asynchronous requests.

items()
Return all of the rows that are in the result set.

Return type list

query
Return a read-only value of the query that was submitted to PostgreSQL.

Return type str

rownumber
Return the current offset of the result set

Return type int

status
Return the status message returned by PostgreSQL after the query was executed.

Return type str

2.5 Connection Pooling

The *PoolManager* class provides top-level access to the queries pooling mechanism, managing pools of connections by DSN in instances of the *Pool* class. The connections are represented by instances of the *Connection* class. *Connection* holds the psycopg2 connection handle as well as lock information that lets the Pool and PoolManager know when connections are busy.

These classes are managed automatically by the *Session* and should rarely be interacted with directly.

If you would like to use the *PoolManager* to shutdown all connections to PostgreSQL, either reference it by class or using the *PoolManager.instance* method.

class queries.pool.**PoolManager**
The connection pool object implements behavior around connections and their use in queries.Session objects.

We carry a pool id instead of the connection URI so that we will not be carrying the URI in memory, creating a possible security issue.

classmethod add(*pid*, *connection*)
Add a new connection and session to a pool.

Parameters

- **pid** (*str*) – The pool id
- **connection** (*psycopg2.extensions.connection*) – The connection to add to the pool

classmethod clean(*pid*)
Clean the specified pool, removing any closed connections or stale locks.

Parameters **pid** (*str*) – The pool id to clean

classmethod create(*pid*, *idle_ttl=60*, *max_size=1*, *time_method=None*)
Create a new pool, with the ability to pass in values to override the default idle TTL and the default maximum size.

A pool's idle TTL defines the amount of time that a pool can be open without any sessions before it is removed.

A pool's max size defines the maximum number of connections that can be added to the pool to prevent unbounded open connections.

Parameters

- **pid** (*str*) – The pool ID
- **idle_ttl** (*int*) – Time in seconds for the idle TTL
- **max_size** (*int*) – The maximum pool size
- **time_method** (*callable*) – Override the use of `time.time()` method for time values.

Raises KeyError

classmethod free(*pid*, *connection*)
Free a connection that was locked by a session

Parameters

- **pid** (*str*) – The pool ID
- **connection** (*psycopg2.extensions.connection*) – The connection to remove

classmethod get(*pid*, *session*)
Get an idle, unused connection from the pool. Once a connection has been retrieved, it will be marked as in-use until it is freed.

Parameters

- **pid** (*str*) – The pool ID
- **session** (`queries.Session`) – The session to assign to the connection

Return type psycopg2.extensions.connection

classmethod get_connection(*pid*, *connection*)
Return the specified *Connection* from the pool.

Parameters

- **pid** (*str*) – The pool ID
- **connection** (*psycopg2.extensions.connection*) – The connection to return for

Return type *queries.pool.Connection*

classmethod has_connection(*pid*, *connection*)
Check to see if a pool has the specified connection

Parameters

- **pid** (*str*) – The pool ID
- **connection** (*psycopg2.extensions.connection*) – The connection to check for

Return type bool

classmethod has_idle_connection(*pid*)
Check to see if a pool has an idle connection

Parameters **pid** (*str*) – The pool ID

Return type bool

classmethod instance()
Only allow a single PoolManager instance to exist, returning the handle for it.

Return type *PoolManager*

classmethod is_full(*pid*)
Return a bool indicating if the specified pool is full

Parameters **pid** (*str*) – The pool id

Return type bool

classmethod lock(*pid*, *connection*, *session*)
Explicitly lock the specified connection in the pool

Parameters

- **pid** (*str*) – The pool id
- **connection** (*psycopg2.extensions.connection*) – The connection to add to the pool
- **session** (queries.Session) – The session to hold the lock

classmethod remove(*pid*)
Remove a pool, closing all connections

Parameters **pid** (*str*) – The pool ID

classmethod remove_connection(*pid*, *connection*)
Remove a connection from the pool, closing it if is open.

Parameters

- **pid** (*str*) – The pool ID
- **connection** (*psycopg2.extensions.connection*) – The connection to remove

Raises ConnectionNotFoundError

classmethod report()
Return the state of the all of the registered pools.

Return type dict

classmethod set_idle_ttl(*pid*, *ttl*)
Set the idle TTL for a pool, after which it will be destroyed.

Parameters

- **pid** (*str*) – The pool id
- **ttl** (*int*) – The TTL for an idle pool

classmethod set_max_size(*pid*, *size*)
Set the maximum number of connections for the specified pool

Parameters

- **pid** (*str*) – The pool to set the size for
- **size** (*int*) – The maximum number of connections

classmethod shutdown()
Close all connections on in all pools

classmethod size(*pid*)
Return the number of connections in the pool

Parameters **pid** (*str*) – The pool id

:rtype int

class queries.pool.**Pool**(*pool_id*, *idle_ttl=60*, *max_size=1*, *time_method=None*)
A connection pool for gaining access to and managing connections

add(*connection*)
Add a new connection to the pool

Parameters **connection** (*psycopg2.extensions.connection*) – The connection to add to the pool

Raises PoolFullError

busy_connections
Return a list of active/busy connections

Return type list

clean()
Clean the pool by removing any closed connections and if the pool's idle has exceeded its idle TTL, remove all connections.

close()
Close the pool by closing and removing all of the connections

closed_connections
Return a list of closed connections

Return type list

connection_handle(*connection*)
Return a connection object for the given psycopg2 connection

Parameters **connection** (*psycopg2.extensions.connection*) – The connection to return a parent for

Return type *Connection*

executing_connections
Return a list of connections actively executing queries

Return type list

free(*connection*)
Free the connection from use by the session that was using it.

Parameters **connection** (*psycopg2.extensions.connection*) – The connection to free

Raises ConnectionNotFoundError

get(*session*)
Return an idle connection and assign the session to the connection

Parameters **session** (queries.Session) – The session to assign

Return type psycopg2.extensions.connection

Raises NoIdleConnectionsError

id
Return the ID for this pool

Return type str

idle_connections
Return a list of idle connections

Return type list

idle_duration
Return the number of seconds that the pool has had no active connections.

Return type float

is_full
Return True if there are no more open slots for connections.

Return type bool

lock(*connection*, *session*)
Explicitly lock the specified connection

Parameters

- **connection** (*psycopg2.extensions.connection*) – The connection to lock
- **session** (queries.Session) – The session to hold the lock

locked_connections
Return a list of all locked connections

Return type list

remove(*connection*)
Remove the connection from the pool

Parameters **connection** (*psycopg2.extensions.connection*) – The connection to remove

Raises ConnectionNotFoundError

Raises ConnectionBusyError

report()
Return a report about the pool state and configuration.

Return type dict

set_idle_ttl(*ttl*)
Set the idle ttl

Parameters **ttl** (*int*) – The TTL when idle

set_max_size(*size*)
Set the maximum number of connections

Parameters **size** (*int*) – The maximum number of connections

shutdown()
Forcefully shutdown the entire pool, closing all non-executing connections.

Raises ConnectionBusyError

class queries.pool.**Connection**(*handle*)

Contains the handle to the connection, the current state of the connection and methods for manipulating the state of the connection.

busy

Return if the connection is currently executing a query or is locked by a session that still exists.

Return type bool

close()

Close the connection

Raises ConnectionBusyError

closed

Return if the psycopg2 connection is closed.

Return type bool

executing

Return if the connection is currently executing a query

Return type bool

free()

Remove the lock on the connection if the connection is not active

Raises ConnectionBusyError

id

Return id of the psycopg2 connection object

Return type int

lock(*session*)

Lock the connection, ensuring that it is not busy and storing a weakref for the session.

Parameters **session** (queries.Session) – The session to lock the connection with

Raises ConnectionBusyError

locked

Return if the connection is currently exclusively locked

Return type bool

2.6 Examples

The following examples show more advanced use of Queries:

2.6.1 Basic TornadoSession Usage

The following example implements a very basic RESTful API. The following DDL will create the table used by the API:

```
CREATE TABLE widgets (sku varchar(10) NOT NULL PRIMARY KEY,
                      name text NOT NULL,
                      qty integer NOT NULL);
```

The Tornado application provides two endpoints: /widget(/sku-value) and /widgets. SKUs are set to be a 10 character value with the regex of `[a-z0-9]{10}`. To add a widget, call PUT on /widget, to update a widget call POST on /widget/[SKU].

```
from tornado import gen, ioloop, web
import queries


class WidgetRequestHandler(web.RequestHandler):
    """Handle the CRUD methods for a widget"""

    def initialize(self):
        """Setup a queries.TornadoSession object to use when the RequestHandler
        is first initialized.

        """
        self.session = queries.TornadoSession()

    def options(self, *args, **kwargs):
        """Let the caller know what methods are supported

        :param list args: URI path arguments passed in by Tornado
        :param list args: URI path keyword arguments passed in by Tornado

        """
        self.set_header('Allow', ', '.join(['DELETE', 'GET', 'POST', 'PUT']))
        self.set_status(204)  # Successful request, but no data returned
        self.finish()

    @gen.coroutine
    def delete(self, *args, **kwargs):
        """Delete a widget from the database

        :param list args: URI path arguments passed in by Tornado
        :param list args: URI path keyword arguments passed in by Tornado

        """
        # We need a SKU, if it wasn't passed in the URL, return an error
        if 'sku' not in kwargs:
            self.set_status(403)
            self.finish({'error': 'missing required value: sku'})

        # Delete the widget from the database by SKU
        else:
            results = yield self.session.query("DELETE FROM widgets WHERE sku=%(sku)s
↪",
                                               {'sku': kwargs['sku']})
            if not results:
                self.set_status(404)
                self.finish({'error': 'SKU not found in system'})
            else:
                self.set_status(204)  # Success, but no data returned
                self.finish()

            # Free the results and release the connection lock from session.query
            results.free()

    @gen.coroutine
```

(continues on next page)

(continued from previous page)

```
    def get(self, *args, **kwargs):
        """Fetch a widget from the database

        :param list args: URI path arguments passed in by Tornado
        :param list args: URI path keyword arguments passed in by Tornado

        """
        # We need a SKU, if it wasn't passed in the URL, return an error
        if 'sku' not in kwargs:
            self.set_status(403)
            self.finish({'error': 'missing required value: sku'})

        # Fetch a row from the database for the SKU
        else:
            results = yield self.session.query("SELECT * FROM widgets WHERE sku=
↪%(sku)s",
                                               {'sku': kwargs['sku']})

            # No rows returned, send a 404 with a JSON error payload
            if not results:
                self.set_status(404)
                self.finish({'error': 'SKU not found in system'})

            # Send back the row as a JSON object
            else:
                self.finish(results.as_dict())

            # Free the results and release the connection lock from session.query
            results.free()

    @gen.coroutine
    def post(self, *args, **kwargs):
        """Update a widget in the database

        :param list args: URI path arguments passed in by Tornado
        :param list args: URI path keyword arguments passed in by Tornado

        """
        # We need a SKU, if it wasn't passed in the URL, return an error
        if 'sku' not in kwargs:
            self.set_status(403)
            self.finish({'error': 'missing required value: sku'})

        # Update the widget in the database by SKU
        else:

            sql = "UPDATE widgets SET name=%(name)s, qty=%(qty)s WHERE sku=%(sku)s"
            try:
                results = yield self.session.query(sql,
                                                   {'sku': kwargs['sku'],
                                                    'name': self.get_argument('name'),
                                                    'qty': self.get_argument('qty')})

                # Free the results and release the connection lock from session.query
                results.free()

            # DataError is raised when there's a problem with the data passed in
```

(continues on next page)

(continued from previous page)

```
            except queries.DataError as error:
                self.set_status(409)
                self.finish({'error': {'error': error.pgerror.split('\n')[0][8:]}})

            else:
                # No rows means there was no record updated
                if not results:
                    self.set_status(404)
                    self.finish({'error': 'SKU not found in system'})

                # The record was updated
                else:
                    self.set_status(204)  # Success, but not returning data
                    self.finish()

    @gen.coroutine
    def put(self, *args, **kwargs):
        """Add a widget to the database

        :param list args: URI path arguments passed in by Tornado
        :param list args: URI path keyword arguments passed in by Tornado

        """
        try:
            results = yield self.session.query("INSERT INTO widgets VALUES (%s, %s,
↪%s)",
                                               [self.get_argument('sku'),
                                                self.get_argument('name'),
                                                self.get_argument('qty')])

            # Free the results and release the connection lock from session.query
            results.free()
        except (queries.DataError,
                queries.IntegrityError) as error:
            self.set_status(409)
            self.finish({'error': {'error': error.pgerror.split('\n')[0][8:]}})
        else:
            self.set_status(201)
            self.finish()


class WidgetsRequestHandler(web.RequestHandler):
    """Return a list of all of the widgets in the database"""

    def initialize(self):
        """Setup a queries.TornadoSession object to use when the RequestHandler
        is first initialized.

        """
        self.session = queries.TornadoSession()

    def options(self, *args, **kwargs):
        """Let the caller know what methods are supported

        :param list args: URI path arguments passed in by Tornado
        :param list args: URI path keyword arguments passed in by Tornado
```

(continues on next page)

(continued from previous page)

```
        """
        self.set_header('Allow', ', '.join(['GET']))
        self.set_status(204)
        self.finish()

    @gen.coroutine
    def get(self, *args, **kwargs):
        """Get a list of all the widgets from the database

        :param list args: URI path arguments passed in by Tornado
        :param list args: URI path keyword arguments passed in by Tornado

        """
        results = yield self.session.query('SELECT * FROM widgets ORDER BY sku')

        # Tornado doesn't allow you to return a list as a JSON result by default
        self.finish({'widgets': results.items()})

        # Free the results and release the connection lock from session.query
        results.free()


if __name__ == "__main__":
    application = web.Application([
        (r"/widget", WidgetRequestHandler),
        (r"/widget/(?P<sku>[a-zA-Z0-9]{10})", WidgetRequestHandler),
        (r"/widgets", WidgetsRequestHandler)
    ]).listen(8888)
    ioloop.IOLoop.instance().start()
```

2.6.2 Concurrent Queries in Tornado

The following example issues multiple concurrent queries in a single asynchronous request and will wait until all queries are complete before progressing:

```
from tornado import gen, ioloop, web
import queries


class RequestHandler(web.RequestHandler):

    def initialize(self):
        self.session = queries.TornadoSession()

    @gen.coroutine
    @gen.coroutine
    def get(self, *args, **kwargs):

        # Issue the three queries and wait for them to finish before progressing
        (q1result,
         q2result,
         q3result) = yield [self.session.query('SELECT * FROM foo'),
                            self.session.query('SELECT * FROM bar'),
                            self.session.query('INSERT INTO requests VALUES (%s, %s,
↪%s)',
```

(continues on next page)

(continued from previous page)

```
                                                [self.remote_ip,
                                                 self.request_uri,
                                                 self.headers.get('User-Agent', '')])]
        # Close the connection
        self.finish({'q1result': q1result.items(),
                     'q2result': q2result.items()})

        # Free the results and connection locks
        q1result.free()
        q2result.free()
        q3result.free()

if __name__ == "__main__":
    application = web.Application([
        (r"/", RequestHandler)
    ]).listen(8888)
    ioloop.IOLoop.instance().start()
```

2.7 Version History

2.7.1 2.0.1 2019-04-04

- Narrow the pin to psycopg2 < 2.8 due to a breaking change
- Fix Results iterator for Python 3.7 (#31 - nvllsvm)

2.7.2 2.0.0 2018-01-29

- REMOVED support for Python 2.6
- FIXED CPU Pegging bug: Cleanup IOLoop and internal stack in `TornadoSession` on connection error. In the case of a connection error, the failure to do this caused CPU to peg @ 100% utilization looping on a non-existent file descriptor. Thanks to cknave for his work on identifying the issue, proposing a fix, and writing a working test case.
- Move the integration tests to use a local docker development environment
- Added new methods `queries.pool.Pool.report` and `queries.pool.PoolManager.Report` for reporting pool status.
- Added new methods to `queries.pool.Pool` for returning a list of busy, closed, executing, and locked connections.

2.7.3 1.10.4 2018-01-10

- Implement `Results.__bool__` to be explicit about Python 3 support.
- Catch any exception raised when using TornadoSession and invoking the execute function in psycopg2 for exceptions raised prior to sending the query to Postgres. This could be psycopg2.Error, IndexError, KeyError, or who knows, it's not documented in psycopg2.

2.7.4 1.10.3 2017-11-01

- Remove the functionality from `TornadoSession.validate` and make it raise a `DeprecationWarning`
- Catch the `KeyError` raised when `PoolManager.clean()` is invoked for a pool that doesn't exist

2.7.5 1.10.2 2017-10-26

- Ensure the pool exists when executing a query in TornadoSession, the new timeout behavior prevented that from happening.

2.7.6 1.10.1 2017-10-24

- Use an absolute time in the call to `add_timeout`

2.7.7 1.10.0 2017-09-27

- Free when tornado_session.Result is `__del__`'d without `free` being called.
- Auto-clean the pool after Results.free TTL+1 in tornado_session.TornadoSession
- Don't raise NotImplementedError in Results.free for synchronous use, just treat as a noop

2.7.8 1.9.1 2016-10-25

- Add better exception handling around connections and getting the logged in user

2.7.9 1.9.0 2016-07-01

- Handle a potential race condition in TornadoSession when too many simultaneous new connections are made and a pool fills up
- Increase logging in various places to be more informative
- Restructure queries specific exceptions to all extend off of a base QueriesException
- Trivial code cleanup

2.7.10 1.8.10 2016-06-14

- Propagate PoolManager exceptions from TornadoSession (#20) - Fix by Dave Shawley

2.7.11 1.8.9 2015-11-11

- Move to psycopg2cffi for PyPy support

2.7.12 1.7.5 2015-09-03

- Don't let Session and TornadoSession share connections

2.7.13 1.7.1 2015-03-25

- Fix TornadoSession's use of cleanup (#8) - Fix by Oren Itamar

2.7.14 1.7.0 2015-01-13

- Implement `Pool.shutdown` and `PoolManager.shutdown` to cleanly shutdown all open, non-executing connections across a Pool or all pools. Update locks in Pool operations to ensure atomicity.

2.7.15 1.6.1 2015-01-09

- Fixes an iteration error when closing a pool (#7) - Fix by Chris McGuire

2.7.16 1.6.0 2014-11-20

- Handle URI encoded password values properly

2.7.17 1.5.0 2014-10-07

- Handle empty query results in the iterator (#4) - Fix by Den Teresh

2.7.18 1.4.0 2014-09-04

- Address exception handling in tornado_session

CHAPTER 3

Issues

Please report any issues to the Github repo at https://github.com/gmr/queries/issues

CHAPTER 4

Source

Queries source is available on Github at https://github.com/gmr/queries

CHAPTER 5

Inspiration

Queries is inspired by Kenneth Reitz's awesome work on requests.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index