Momoko Documentation

Release 1.1.6

Frank Smit

May 02, 2015

Contents

Momoko wraps Psycopg2's functionality for use in Tornado.

The latest source code can be found on Github and bug reports can be sent there too. All releases will be uploaded to PyPi.

Contents:

CHAPTER 1

Changelog

1.1 1.1.6 (2015-04-26)

- Aadded register_json
- Docs: fix typos, spelling, grammatical errors; improve unclear wording
- Removed support for psycopg2ct

1.2 1.1.5 (2014-11-17)

- Catching ALL types of early error. Fixes issue 79.

1.3 1.1.4 (2014-07-21)

- Tornado 4.0 compatablity: backported old `Task` class for Tornado 4.0 compatablity.

1.4 1.1.3 (2014-05-21)

- Fixed hstore.

1.5 1.1.2 (2014-03-06)

- Fixed a minor Python 3.2 issue.

1.6 1.1.1 (2014-03-06)

Fixes:

- `Connection.transaction` does not break when passed SQL strings are of `unicode` type

1.7 1.1.0 (2014-02-24)

New features:

- Transparent automatic reconnects if database disappears and comes back.
- Session init commands (`setsession`).
- Dynamic pool size stretching. New connections will be opened under load up-to predefined limit.
- API for manual connection management with `getconn`/`putconn`. Useful for server-side cursors.
- A lot of internal improvements and cleanup.

Fixes:

- Connections are managed explicitly - eliminates transaction problems reported.
- `connection_factory` (and `curosr_factor`) arguments handled properly by `Pool`.

1.8 1.0.0 (2013-05-01)

- Fix code example in documentation. By matheuspatury in pull request 46

1.9 1.0.0b2 (2013-02-28)

- Tested on CPython 2.6, 2.7, 3.2, 3.3 and PyPy with Psycopg2, psycopg2ct and psycopg2cffi.
- Add and remove a database connection to and from the IOLoop for each operation. See pull request 38 and commits 189323211b and 92940db0a0 for more information.
- Replaced dynamic connection pool with a static one.
- Add support for hstore.

1.10 1.0.0b1 (2012-12-16)

This is a beta release. It means that the code has not been tested thoroughly yet. This first beta release is meant to provide all the functionality of the previous version plus a few additions.

- Most of the code has been rewritten.
- The mogrify method has been added.
- Added support for transactions.
- The query chain and batch have been removed, because `tornado.gen` can be used instead.
- Error reporting has bee improved by passing the raised exception to the callback. A callback accepts two arguments: the cursor and the error.
- `Op`, `WaitOp` and `WaitAllOps` in `momoko.utils` are wrappers for classes in `tornado.gen` which raise the error again when one occurs. And the user can capture the exception in the request handler.
- A complete set of tests has been added in the `momoko` module: `momoko.tests`. These can be run with `python setup.py test`.

1.11 0.5.0 (2012-07-30)

- Removed all Adisp related code.
- Refactored connection pool and connection polling.
- Just pass all unspecified arguments to `BlockingPool` and `AsyncPool`. So `connection_factory` can be used again.

1.12 0.4.0 (2011-12-15)

- Reorganized classes and files.
- Renamed `momoko.Client` to `momoko.AsyncClient`.
- Renamed `momoko.Pool` to `momoko.AsyncPool`.
- Added a client and pool for blocking connections, `momoko.BlockingClient` and `momoko.BlockingPool`.
- Added `PoolError` to the import list in `__init__.py`.
- Added an example that uses Tornado's gen module and Swift.
- Callbacks are now optional for `AsyncClient`.
- `AsyncPool` and `Poller` now accept a ioloop argument. [fzzbt]
- Unit tests have been added. [fzzbt]

1.13 0.3.0 (2011-08-07)

- Renamed `momoko.Momoko` to `momoko.Client`.
- Programming in blocking-style is now possible with `AdispClient`.
- Support for Python 3 has been added.
- The batch and chain fucntion now accepts different arguments. See the documentation for details.

1.14 0.2.0 (2011-04-30)

- Removed `executemany` from `Momoko`, because it can not be used in asynchronous mode.
- Added a wrapper class, `Momoko`, for `Pool`, `BatchQuery` and `QueryChain`.
- Added the `QueryChain` class for executing a chain of queries (and callables) in a certain order.
- Added the `BatchQuery` class for executing batches of queries at the same time.
- Improved `Pool._clean_pool`. It threw an `IndexError` when more than one connection needed to be closed.

1.15 0.1.0 (2011-03-13)

- Initial release.

CHAPTER 2

Installation

Momoko supports Python 2 and 3 and PyPy with psycopg2cffi. And the only dependencies are Tornado and Psycopg2 (or psycopg2cffi). Installation is easy using *easy_install* or pip:

```
pip install momoko
```

The lastest source code can always be cloned from the Github repository with:

```
git clone git://github.com/FSX/momoko.git
cd momoko
python setup.py install
```

Psycopg2 is used by default when installing Momoko, but psycopg2cffi can also be used by setting the `MOMOKO_PSYCOPG2_IMPL` environment variable to `psycopg2cffi` before running `setup.py`. For example:

```
# 'psycopg2' or 'psycopg2cffi'
export MOMOKO_PSYCOPG2_IMPL='psycopg2cffi'
```

The unit tests als use this variable. It needs to be set if something else is used instead of Psycopg2 when running the unit tests. Besides `MOMOKO_PSYCOPG2_IMPL` there are also other variables that need to be set for the unit tests.

Here's an example for the environment variables:

```
export MOMOKO_TEST_DB='your_db'  # Default: momoko_test
export MOMOKO_TEST_USER='your_user'  # Default: postgres
export MOMOKO_TEST_PASSWORD='your_password'  # Empty de default
export MOMOKO_TEST_HOST='localhost'  # Empty de default
export MOMOKO_TEST_PORT='5432'  # Default: 5432

# Set to '0' if hstore extension isn't enabled
export MOMOKO_TEST_HSTORE='1'  # Default: 0

# Set to '0' if json data type isn't enabled (PostgreSQL 9.1 and earlier)
export MOMOKO_TEST_JSON='1'  # Default: 0
```

And running the tests is easy:

```
python setup.py test
```

CHAPTER 3

Tutorial

This tutorial will demonstrate all the functionality found in Momoko. It's assumed a working PostgreSQL database is available, and everything is done in the context of a simple tornado web application. Not everything is explained: because Momoko just wraps Psycopg2, the Psycopg2 documentation must be used alongside Momoko's.

3.1 Boilerplate

Here's the code that's needed for this tutorial. Each example will replace parts or extend upon this code. The code is kept simple and minimal; its purpose is just to demonstrate Momoko's functionality. Here it goes:

```
from tornado import gen
from tornado.ioloop import IOLoop
from tornado.httpserver import HTTPServer
from tornado.options import parse_command_line
from tornado.web import *

import psycopg2
import momoko


class BaseHandler(RequestHandler):
    @property
    def db(self):
        return self.application.db


class TutorialHandler(BaseHandler):
    def get(self):
        self.write('Some text here!')
        self.finish()


if __name__ == '__main__':
    parse_command_line()
    application = Application([
        (r'/', TutorialHandler)
    ], debug=True)

    application.db = momoko.Pool(
        dsn='dbname=your_db user=your_user password=very_secret_password '
            'host=localhost port=5432',
```

```
        size=1
    )

    http_server = HTTPServer(application)
    http_server.listen(8888, 'localhost')
    IOLoop.instance().start()
```

For more information about all the parameters passed to `momoko.Pool` see `momoko.Pool` in the API documentation.

3.2 Usage

`execute()`, `callproc()`, `transaction()` and `mogrify()` are methods of `momoko.Pool` which can be used to query the database. (Actually, `mogrify()` is only used to escape strings, but it needs a connection). All these methods, except `mogrify()`, return a cursor or an exception object. All of the described retrieval methods in Psycopg2's documentation—fetchone, fetchmany, fetchall, etc.—can be used to fetch the results.

All of the example will be using tornado.gen instead of callbacks, because callbacks are fairly simple and don't require as much explanation. Here's one example using a callback:

```
class TutorialHandler(BaseHandler):
    @asynchronous
    def get(self):
        self.db.execute('SELECT 1;', callback=self._done)

    def _done(self, cursor, error):
        self.write('Results: %r' % (cursor.fetchall(),))
        self.finish()
```

The callback only needs to accept two parameters: the cursor and an exception object. The exception object is either `None` or an instance of one of Psycopg2's exceptions. That's all there's to know when using callbacks.

Instead of using tornado.gen directly (or using plain callbacks) Momoko provides subclasses of Task, Wait and WaitAll that have some advantages. These are `Op`, `WaitOp` and `WaitAllOps`. These three classes yield only a cursor and raise an exception when something goes wrong. Here's an example using `Op`:

```
class TutorialHandler(BaseHandler):
    @gen.coroutine
    def get(self):
        try:
            cursor = yield momoko.Op(self.db.execute, 'SELECT 1;')
        except (psycopg2.Warning, psycopg2.Error) as error:
            self.write(str(error))
        else:
            self.write('Results: %r' % (cursor.fetchall(),))

        self.finish()
```

An example with `WaitOp`:

```
class TutorialHandler(BaseHandler):
    @gen.coroutine
    def get(self):
        self.db.execute('SELECT 1;', callback=(yield gen.Callback('q1')))
        self.db.execute('SELECT 2;', callback=(yield gen.Callback('q2')))
        self.db.execute('SELECT 3;', callback=(yield gen.Callback('q3')))
```

```
        try:
            cursor1 = yield momoko.WaitOp('q1')
            cursor2 = yield momoko.WaitOp('q2')
            cursor3 = yield momoko.WaitOp('q3')
        except (psycopg2.Warning, psycopg2.Error) as error:
            self.write(str(error))
        else:
            self.write('Q1: %r<br>' % (cursor1.fetchall(),))
            self.write('Q2: %r<br>' % (cursor2.fetchall(),))
            self.write('Q3: %r<br>' % (cursor3.fetchall(),))

        self.finish()
```

`WaitAllOps` can be used instead of three separate `WaitOp` calls:

```
try:
    cursor1, cursor2, cursor3 = yield momoko.WaitAllOps(('q1', 'q2', 'q3'))
except (psycopg2.Warning, psycopg2.Error) as error:
    self.write(str(error))
else:
    self.write('Q1: %r<br>' % (cursor1.fetchall(),))
    self.write('Q2: %r<br>' % (cursor2.fetchall(),))
    self.write('Q3: %r<br>' % (cursor3.fetchall(),))
```

All the above examples use `execute()`, but work with `callproc()`, `transaction()` and `mogrify()` too.

3.3 Advanced

3.3.1 Manual connection management

You can manually acquire connection from the pool using the `getconn()` method. This is very useful, for example, for server-side cursors.

It important to return connection back to the pool once you've done with it, even if an error occurs in the middle of your work. Use either `putconn()` method or `manage()` manager to return the connection.

Here is the server-side cursor example (based on the code in momoko unittests):

```
@gen.coroutine
def get(self):
    chunk = 1000
    try:
        connection = yield momoko.Op(self.db.getconn)
        with self.db.manage(connection):
            yield momoko.Op(connection.execute, "BEGIN")
            yield momoko.Op(connection.execute, "DECLARE all_ints CURSOR FOR SELECT * FROM unit_test_
            rows = True
            while rows:
                cursor = yield momoko.Op(connection.execute, "FETCH %s FROM all_ints", (chunk,))
                rows = cursor.fetchall()
                # Do something with results...
            yield momoko.Op(connection.execute, "CLOSE all_ints")
            yield momoko.Op(connection.execute, "COMMIT")
    except Exception as error:
        self.write(str(error))
```

CHAPTER 4

API

Classes, methods and stuff.

4.1 Connections

class momoko.**Pool**(*dsn*, *connection_factory=None*, *cursor_factory=None*, *size=1*, *max_size=None*, *callback=None*, *ioloop=None*, *raise_connect_errors=True*, *reconnect_interval=500*, *setsession=[]*)

Asynchronous connection pool.

The pool manages database connections and passes operations to connections.

See momoko.Connection for documentation about the dsn, connection_factory and cursor_factory parameters. These are used by the connection pool when a new connection is created.

Parameters

- **size** (*integer*) – Amount of connections created upon initialization. Defaults to 1.
- **max_size** (*integer*) – Allow number of connection to grow under load up to given size. Defaults to size.
- **callback** (*callable*) – A callable that's called after all the connections are created. Defaults to None.
- **ioloop** – An instance of Tornado's IOLoop. Defaults to None, IOLoop.instance() will be used.
- **raise_connect_errors** (*bool*) – Whether to raise exception if database connection fails. Set to False to enable automatic reconnection attempts. Defaults to True.
- **reconnect_interval** (*integer*) – When using automatic reconnects, set minimum reconnect interval, in milliseconds, before retrying connection attempt. Don't set this value too low to prevent "banging" the database server with connection attempts. Defaults to 500.
- **setsession** (*list*) – List of intial sql commands to be executed once connection is established. If any of the commands failes, the connection will be closed. **NOTE:** The commands will be executed as one transaction block.

callproc(*procname*, *parameters=()*, *cursor_factory=None*, *callback=None*)

Call a stored database procedure with the given name.

See momoko.Connection.callproc() for documentation about the parameters.

close()
Close the connection pool.

execute(*operation*, *parameters=()*, *cursor_factory=None*, *callback=None*)
Prepare and execute a database operation (query or command).

See `momoko.Connection.execute()` for documentation about the parameters.

getconn(*ping=True*, *callback=None*)
Acquire connection from the pool.

You can then use this connection for subsequest queries. Just supply, for example, `connection.execute` instead of `Pool.execute` to `momoko.Op`.

Make sure to return connection to the pool by calling `momoko.Pool.putconn()`, otherwise the connection will remain forever-busy and you'll starvate your pool quickly.

Parameters **ping** (*boolean*) – Whether to ping connection before returning it by executing `momoko.Pool.ping()`.

manage(**args*, ***kwds*)
Context manager that automatically returns connection to the pool. You can use it instead of `momoko.Pool.putconn()`:

```
connection = yield momoko.Op(self.db.getconn)
with self.db.manage(connection):
    cursor = yield momoko.Op(connection.execute, "BEGIN")
    ...
```

mogrify(*operation*, *parameters=()*, *callback=None*)
Return a query string after arguments binding.

See `momoko.Connection.mogrify()` for documentation about the parameters.

ping(*connection*, *callback=None*)
Ping given connection object to make sure its alive (involves roundtrip to the database server).

See `momoko.Connection.ping()` for documentation about the details.

putconn(*connection*)
Retrun busy connection back to the pool.

Parameters **connection** (*Connection*) – Connection object previously returned by `momoko.Pool.getconn()`. **NOTE:** This is a synchronous function

register_hstore(*unicode=False*, *callback=None*)
Register adapter and typecaster for `dict-hstore` conversions.

See `momoko.Connection.register_hstore()` for documentation about the parameters. This method has no `globally` parameter, because it already registers hstore to all the connections in the pool.

register_json(*loads=None*, *callback=None*)
Register adapter and typecaster for `dict-json` conversions.

See `momoko.Connection.register_json()` for documentation about the parameters. This method has no `globally` parameter, because it already registers json to all the connections in the pool.

transaction(*statements*, *cursor_factory=None*, *callback=None*)
Run a sequence of SQL queries in a database transaction.

See `momoko.Connection.transaction()` for documentation about the parameters.

class `momoko.`**Connection**
Initiate an asynchronous connect.

Parameters

- **dsn** (*string*) – A Data Source Name string containing one of the following values:
  - **dbname** - the database name
  - **user** - user name used to authenticate
  - **password** - password used to authenticate
  - **host** - database host address (defaults to UNIX socket if not provided)
  - **port** - connection port number (defaults to 5432 if not provided)

  Or any other parameter supported by PostgreSQL. See the PostgreSQL documentation for a complete list of supported parameters.
- **connection_factory** – The `connection_factory` argument can be used to create non-standard connections. The class returned should be a subclass of psycopg2.extensions.connection. See Connection and cursor factories for details. Defaults to `None`.
- **cursor_factory** – The `cursor_factory` argument can be used to return non-standart cursor class The class returned should be a subclass of psycopg2.extensions.cursor. See Connection and cursor factories for details. Defaults to `None`.
- **callback** (*callable*) – A callable that's called after the connection is created. It accepts one paramater: an instance of `momoko.Connection`. Defaults to `None`.
- **ioloop** – An instance of Tornado's IOLoop. Defaults to `None`.
- **setsession** (*list*) – List of intial sql commands to be executed once connection is established. If any of the commands failes, the connection will be closed. **NOTE:** The commands will be executed as one transaction block.

busy()

(Deprecated) Check if the connection is busy or not.

callproc(**args*, ***kwargs*)

Call a stored database procedure with the given name.

The sequence of parameters must contain one entry for each argument that the procedure expects. The result of the call is returned as modified copy of the input sequence. Input parameters are left untouched, output and input/output parameters replaced with possibly new values.

The procedure may also provide a result set as output. This must then be made available through the standard fetch*() methods.

Parameters

- **procname** (*string*) – The name of the database procedure.
- **parameters** (*tuple/list*) – A list or tuple with query parameters. See Passing parameters to SQL queries for more information. Defaults to an empty tuple.
- **cursor_factory** – The `cursor_factory` argument can be used to create non-standard cursors. The class returned must be a subclass of psycopg2.extensions.cursor. See Connection and cursor factories for details. Defaults to `None`.
- **callback** (*callable*) – A callable that is executed when the query has finished. It must accept two positional parameters. The first one being the cursor and the second one `None` or an instance of an exception if an error has occurred, in that case the first parameter will be `None`. Defaults to `None`. **NOTE:** *callback* should always passed as keyword argument

close()
Remove the connection from the IO loop and close it.

closed
Indicates whether the connection is closed or not.

execute(**args*, ***kwargs*)
Prepare and execute a database operation (query or command).

Parameters

- **operation** (*string*) – An SQL query.
- **parameters** (*tuple/list*) – A list or tuple with query parameters. See Passing parameters to SQL queries for more information. Defaults to an empty tuple.
- **cursor_factory** – The `cursor_factory` argument can be used to create non-standard cursors. The class returned must be a subclass of psycopg2.extensions.cursor. See Connection and cursor factories for details. Defaults to `None`.
- **callback** (*callable*) – A callable that is executed when the query has finished. It must accept two positional parameters. The first one being the cursor and the second one `None` or an instance of an exception if an error has occurred, in that case the first parameter will be `None`. Defaults to `None`. **NOTE:** *callback* should always passed as keyword argument

mogrify(**args*, ***kwargs*)
Return a query string after arguments binding.

The string returned is exactly the one that would be sent to the database running the execute() method or similar.

Parameters

- **operation** (*string*) – An SQL query.
- **parameters** (*tuple/list*) – A list or tuple with query parameters. See Passing parameters to SQL queries for more information. Defaults to an empty tuple.
- **callback** (*callable*) – A callable that is executed when the query has finished. It must accept two positional parameters. The first one being the resulting query as a byte string and the second one `None` or an instance of an exception if an error has occurred. Defaults to `None`. **NOTE:** *callback* should always passed as keyword argument

ping(**args*, ***kwargs*)
Make sure this connection is alive by executing SELECT 1 statement - i.e. roundtrip to the database.

NOTE: On the contrary to other methods, callback function signature is `callback(self, error)` and not `callback(cursor, error)`.

NOTE: *callback* should always passed as keyword argument

register_hstore(*globally=False*, *unicode=False*, *callback=None*)
Register adapter and typecaster for `dict-hstore` conversions.

More information on the hstore datatype can be found on the Psycopg2 documentation.

Parameters

- **globally** (*boolean*) – Register the adapter globally, not only on this connection.
- **unicode** (*boolean*) – If `True`, keys and values returned from the database will be `unicode` instead of `str`. The option is not available on Python 3.

NOTE: *callback* should always passed as keyword argument

register_json(**args*, ***kwargs*)
Register adapter and typecaster for `dict-json` conversions.

More information on the json datatype can be found on the Psycopg2 documentation.

Parameters

- **globally** (*boolean*) – Register the adapter globally, not only on this connection.
- **loads** (*function*) – The function used to parse the data into a Python object. If `None` use `json.loads()`, where `json` is the module chosen according to the Python version. See psycopg2.extra docs.

NOTE: *callback* should always passed as keyword argument

transaction(*statements*, *cursor_factory=None*, *callback=None*)
Run a sequence of SQL queries in a database transaction.

Parameters

- **statements** (*tuple/list*) – List or tuple containing SQL queries with or without parameters. An item can be a string (SQL query without parameters) or a tuple/list with two items, an SQL query and a tuple/list wuth parameters.

  See Passing parameters to SQL queries for more information.
- **cursor_factory** – The `cursor_factory` argument can be used to create non-standard cursors. The class returned must be a subclass of psycopg2.extensions.cursor. See Connection and cursor factories for details. Defaults to `None`.
- **callback** (*callable*) – A callable that is executed when the transaction has finished. It must accept two positional parameters. The first one being a list of cursors in the same order as the given statements and the second one `None` or an instance of an exception if an error has occurred, in that case the first parameter is an empty list. Defaults to `None`. **NOTE:** *callback* should always passed as keyword argument

4.2 Utilities

class momoko.**Op**(*func*, **args*, ***kwargs*)
Run a single asynchronous operation.

Behaves like tornado.gen.Task, but raises an exception (one of Psycop2's exceptions) when an error occurs related to Psycopg2 or PostgreSQL.

class momoko.**WaitOp**(*key*)
Return the argument passed to the result of a previous tornado.gen.Callback.

Behaves like tornado.gen.Wait, but raises an exception (one of Psycop2's exceptions) when an error occurs related to Psycopg2 or PostgreSQL.

class momoko.**WaitAllOps**(*keys*)
Return the results of multiple previous tornado.gen.Callback.

Behaves like tornado.gen.WaitAll, but raises an exception (one of Psycop2's exceptions) when an error occurs related to Psycopg2 or PostgreSQL.

4.3 Exceptions

class momoko.**PoolError**

The `PoolError` exception is raised when something goes wrong in the connection pool. When the maximum amount is exceeded for example.

CHAPTER 5

Indices and tables

- *genindex*
- *search*

m

B

C

E

G

M

O

P

R

T

W