Momoko Documentation

Release 2.2.1

Frank Smit

October 13, 2015

Contents

1 Changelog **3**
1.1 2.2.1 (2015-10-13) . . . 3
1.2 2.2.0 (2015-09-20) . . . 3
1.3 2.1.1 (2015-08-03) . . . 3
1.4 2.1.0 (2015-07-08) . . . 3
1.5 2.0.0 (2015-05-10) . . . 3
1.6 1.1.6 (2015-04-26) . . . 3
1.7 1.1.5 (2014-11-17) . . . 4
1.8 1.1.4 (2014-07-21) . . . 4
1.9 1.1.3 (2014-05-21) . . . 4
1.10 1.1.2 (2014-03-06) . . . 4
1.11 1.1.1 (2014-03-06) . . . 4
1.12 1.1.0 (2014-02-24) . . . 4
1.13 1.0.0 (2013-05-01) . . . 4
1.14 1.0.0b2 (2013-02-28) . . . 5
1.15 1.0.0b1 (2012-12-16) . . . 5
1.16 0.5.0 (2012-07-30) . . . 5
1.17 0.4.0 (2011-12-15) . . . 5
1.18 0.3.0 (2011-08-07) . . . 6
1.19 0.2.0 (2011-04-30) . . . 6
1.20 0.1.0 (2011-03-13) . . . 6

2 Installation **7**

3 Tutorial **9**
3.1 The principle . . . 9
3.2 Trival example . . . 9
3.3 Introducing Pool . . . 10
3.4 Using Pool . . . 11
3.5 Advanced . . . 12

4 API **13**
4.1 Connections . . . 13
4.2 Exceptions . . . 18

5 Indices and tables **19**

Python Module Index **21**

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 2.2.1 (2015-10-13)

- Wait for pending connections during connection acquiring (issue 122). Thanks to jbowes.

1.2 2.2.0 (2015-09-20)

- Fixed serious flaw with connection retrials. More details.
- Fixed ping to handle failures properly (issue 115).
- NOTE: gcc is now required to run tests - we supply built-in version of tcproxy for connection failure simulation.

1.3 2.1.1 (2015-08-03)

- Fixed JSON/HSTORE support with named cursors (issue 112). Thanks to helminster.

1.4 2.1.0 (2015-07-08)

- Auto shrink support. Thanks to John Chumnanvech.

1.5 2.0.0 (2015-05-10)

- Full rewrite using using Futures
- NOTE: The new API is similar but not backwards compatible. Make sure to read documentation first.

1.6 1.1.6 (2015-04-26)

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

1.7 1.1.5 (2014-11-17)

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

1.8 1.1.4 (2014-07-21)

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

1.9 1.1.3 (2014-05-21)

- Fixed hstore.

1.10 1.1.2 (2014-03-06)

- Fixed a minor Python 3.2 issue.

1.11 1.1.1 (2014-03-06)

Fixes:

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

1.12 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`.

1.13 1.0.0 (2013-05-01)

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

1.14 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.

1.15 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`.

1.16 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.

1.17 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]

1.18 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.

1.19 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.

1.20 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 all 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
```

Momoko tests use tcproxy for simulating Postgres server unavailablity. The copy of tcproxy is bundled with Momoko, but you need to build it first:

```
make -C tcproxy
```

Finally, 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 The principle

All of the `Pool()` and `Connection()` methods return _futures. There are some notable exceptions like `close()` - make sure to consule API documentation for the details.

These future objects can be simply `yield`-ed in Tornado methods decorated with `gen.coroutine`. For SQL execution related methods these futures resolve to corresponding cursor objects.

3.2 Trival example

Here is the simplest synchronous version of connect/select code:

```
import psycopg2
conn = psycopg2.connect(dsn="...")
cursor = conn.cursor()
cursor.execute("SELECT 1")
rows = cursor.fetchall()
```

And this is how the same code looks with Momoko/Tornado:

```
import momoko
from tornado.ioloop import IOLoop
ioloop = IOLoop.instance()

conn = momoko.Connection(dsn="...")
future = ioloop.add_future(conn.connect(), lambda x: ioloop.stop())
ioloop.start()
future.result()  # raises exception on connection error

future = conn.execute("SELECT 1")
ioloop.add_future(future, lambda x: ioloop.stop())
ioloop.start()
cursor = future.result()
rows = cursor.fetchall()
```

We create connection object. Then invoke `connect()` method that returns future that resolves to connection object itself when connection is ready (we already have connection object at hand, thus we just wait until future is ready, ignoring its result).

Next we call `execute()` which returns future that resolves to ready-to-use cursor object. And we use IOLoop again to wait for this future to be ready.

Now you know to use *`Connection()`* for working with with stand-alone connections to PostgreSQL in asynchronous mode.

3.3 Introducing Pool

The real power of Momoko comes with *`Pool()`*. It provides several nice features that make it useful in production environments:

Connection pooling It manages several connections and distributes queries requests between them. If all connections are busy, outstanding query requests are waiting in queue

Automatic pool growing (stretching) You can allow automatic stretching - i.e. if all connections are busy and more requests are coming, Pool will open more connections up a certain limit

Automatic reconnects If connections get terminated (database server restart, etc) Pool will automatically reconnect them and transparently retry query if it failed due to dead connection.

3.3.1 Boilerplate

Here's the code that's needed for the rest of 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 import web

import psycopg2
import momoko


class BaseHandler(web.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 = web.Application([
        (r'/', TutorialHandler)
    ], debug=True)
```

```
    ioloop = IOLoop.instance()

    application.db = momoko.Pool(
        dsn='dbname=your_db user=your_user password=very_secret_password '
            'host=localhost port=5432',
        size=1,
        ioloop=ioloop,
    )

    # this is a one way to run ioloop in sync
    future = application.db.connect()
    ioloop.add_future(future, lambda f: ioloop.stop())
    ioloop.start()
    future.result()  # raises exception on connection error

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

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

3.4 Using Pool

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.

First, lets rewrite our trivial example using Tornado web handlers:

```
class TutorialHandler(BaseHandler):
    @gen.coroutine
    def get(self):
        cursor = yield self.db.execute("SELECT 1;")
        self.write("Results: %s" % cursor.fetchone())
        self.finish()
```

To execute several queries in parallel, accumulate corresponding futures and yield them at once:

```
class TutorialHandler(BaseHandler):
    @gen.coroutine
    def get(self):
        try:
            f1 = self.db.execute('select 1;')
            f2 = self.db.execute('select 2;')
            f3 = self.db.execute('select 3;')
            yield [f1, f2, f3]

            cursor1 = f1.result()
            cursor2 = f2.result()
            cursor3 = f3.result()

        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()
```

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

3.5 Advanced

3.5.1 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):
    int_count = 1000
    offset = 0
    chunk = 10
    try:
        conn = yield self.db.getconn()
        with self.db.manage(conn):
            yield conn.execute("BEGIN")
            yield conn.execute("DECLARE all_ints CURSOR FOR SELECT * FROM unit_test_int_table")
            while offset < int_count:
                cursor = yield conn.execute("FETCH %s FROM all_ints", (chunk,))
                rows = cursor.fetchall()
                # Do something with results...
                offset += chunk
            yield conn.execute("CLOSE all_ints")
            yield conn.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*, *ioloop=None*, *raise_connect_errors=True*, *reconnect_interval=500*, *setsession=()*, *auto_shrink=False*, *shrink_delay=datetime.timedelta(0, 120)*, *shrink_period=datetime.timedelta(0, 120)*)

Asynchronous conntion pool object. All its methods are asynchronous unless stated otherwide in method description.

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`.
- **size** (*int*) – Minimal number of connections to maintain. `size` connections will be opened and maintained after calling *`momoko.Pool.connect()`*.
- **max_size** (*int or None*) – if not `None`, the pool size will dynamically grow on demand up to `max_size` open connections. By default the connections will still be maintained even if when the pool load decreases. See also `auto_shrink` parameter.

- **ioloop** – Tornado IOloop instance to use. Defaults to Tornado's `IOLoop.instance()`.
- **raise_connect_errors** (*bool*) – Whether to raise `momoko.PartiallyConnectedError()` when failing to connect to database during `momoko.Pool.connect()`.
- **reconnect_interval** (*int*) – If database server becomes unavailble, the pool will try to reestablish the connection. The attempt frequency is `reconnect_interval` milliseconds.
- **setsession** (*list*) – List of intial sql commands to be executed once connection is established. If any of the commands fails, the connection will be closed. **NOTE:** The commands will be executed as one transaction block.
- **auto_shrink** (*bool*) – Garbage-collect idle connections. Only applicable if `max_size` was specified. Nevertheless, the pool will mainatain at least `size` connections.
- **shrink_delay** (`datetime.timedelta()`) – A connection is declared idle if it was not used for `shrink_delay` time period. Idle connections will be garbage-collected if `auto_shrink` is set to `True`.
- **shrink_period** (`datetime.timedelta()`) – If `auto_shink` is enabled, this parameter defines how the pool will check for idle connections.

exception DatabaseNotAvailable

Raised when Pool can not connect to database server

`Pool.`**callproc**(**args*, ***kwargs*)

Call a stored database procedure with the given name.

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

`Pool.`**close**()

Close the connection pool.

NOTE: This is a synchronous method.

`Pool.`**connect**()

Returns future that resolves to this Pool object.

If some connection failed to connect *and* self.raise_connect_errors is true, raises `momoko.PartiallyConnectedError()`.

`Pool.`**execute**(**args*, ***kwargs*)

Prepare and execute a database operation (query or command).

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

`Pool.`**getconn**(*ping=True*)

Acquire connection from the pool.

You can then use this connection for subsequest queries. Just use `connection.execute` instead of `Pool.execute`.

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.

Returns future that resolves to the acquired connection object.

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

`Pool.manage(*args, **kwds)`

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

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

`Pool.mogrify(*args, **kwargs)`

Return a query string after arguments binding.

NOTE: This is NOT a synchronous method (contary to *momoko.Connection.mogrify*) - it asynchronously waits for available connection. For performance reasons, its better to create dedicated `momoko.Connection()` object and use it directly for mogrification, this operation does not imply any real operation on the database server.

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

`Pool.ping()`

Make sure this connection is alive by executing SELECT 1 statement - i.e. roundtrip to the database.

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

`Pool.putconn(connection)`

Return busy connection back to the pool.

NOTE: This is a synchronous method.

Parameters **connection** (Connection) – Connection object previously returned by `momoko.Pool.getconn()`.

`Pool.register_hstore(*args, **kwargs)`

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.

`Pool.register_json(*args, **kwargs)`

Create and register typecasters converting `json` type to Python objects.

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.

`Pool.transaction(*args, **kwargs)`

Run a sequence of SQL queries in a database transaction.

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

class `momoko.Connection(dsn, connection_factory=None, cursor_factory=None, ioloop=None, setsession=())`

Asynchronous connection object. All its methods are asynchronous unless stated otherwide in method description.

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`.
- **setsession** (*list*) – List of intial sql commands to be executed once connection is established. If any of the commands fails, the connection will be closed. **NOTE:** The commands will be executed as one transaction block.

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

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`.

Returns future that resolves to cursor object containing result.

close ()

Closes the connection.

NOTE: This is a synchronous method.

closed

Indicates whether the connection is closed or not.

connect ()

Initiate asynchronous connect. Returns future that resolves to this connection object.

execute (*operation*, *parameters=()*, *cursor_factory=None*)

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`.

Returns future that resolves to cursor object containing result.

mogrify(*operation*, *parameters=()*)

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.

NOTE: This is a synchronous method.

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.

ping()

Make sure this connection is alive by executing SELECT 1 statement - i.e. roundtrip to the database.

Returns future. If it resolves sucessfully - the connection is alive (or dead otherwise).

register_hstore(*globally=False*, *unicode=False*)

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.

Returns future that resolves to `None`.

register_json(*globally=False*, *loads=None*)

Create and register typecasters converting `json` type to Python objects.

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.

Returns future that resolves to `None`.

transaction(*statements*, *cursor_factory=None*, *auto_rollback=True*)

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`.
- **auto_rollback** (*bool*) – If one of the transaction statements fails, try to automatically execute ROLLBACK to abort the transaction. If ROLLBACK fails, it would not be raised, but only logged.

Returns future that resolves to `list` of cursors. Each cursor contains the result of the corresponding transaction statement.

`momoko.`**`connect`**(**args*, ***kwargs*)

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

Returns future that resolves to `momoko.Connection()` object or raises exception.

4.2 Exceptions

class `momoko.`**`PoolError`**

Raised when something goes wrong in the connection pool.

class `momoko.`**`PartiallyConnectedError`**

Raised when `momoko.Pool()` can not initialize all of the requested connections.

CHAPTER 5

Indices and tables

- genindex
- search

m

momoko, 12

C

callproc() (momoko.Connection method), 16
callproc() (momoko.Pool method), 14
close() (momoko.Connection method), 16
close() (momoko.Pool method), 14
closed (momoko.Connection attribute), 16
connect() (in module momoko), 18
connect() (momoko.Connection method), 16
connect() (momoko.Pool method), 14
Connection (class in momoko), 15

E

execute() (momoko.Connection method), 16
execute() (momoko.Pool method), 14

G

getconn() (momoko.Pool method), 14

M

manage() (momoko.Pool method), 14
mogrify() (momoko.Connection method), 17
mogrify() (momoko.Pool method), 15
momoko (module), 12

P

PartiallyConnectedError (class in momoko), 18
ping() (momoko.Connection method), 17
ping() (momoko.Pool method), 15
Pool (class in momoko), 13
Pool.DatabaseNotAvailable, 14
PoolError (class in momoko), 18
putconn() (momoko.Pool method), 15

R

register_hstore() (momoko.Connection method), 17
register_hstore() (momoko.Pool method), 15
register_json() (momoko.Connection method), 17
register_json() (momoko.Pool method), 15

T

transaction() (momoko.Connection method), 17
transaction() (momoko.Pool method), 15