# txpostgres Documentation

***Release 1.5.0***

**Jan Urbański**

October 13, 2015

Contents

CHAPTER 1

# Module usage

Basic usage of the module is not very different from using Twisted's adbapi:

```
from txpostgres import txpostgres

from twisted.internet import reactor
from twisted.python import log, util

# connect to the database
conn = txpostgres.Connection()
d = conn.connect('dbname=postgres')

# run the query and print the result
d.addCallback(lambda _: conn.runQuery('select tablename from pg_tables'))
d.addCallback(lambda result: util.println('All tables:', result))

# close the connection, log any errors and stop the reactor
d.addCallback(lambda _: conn.close())
d.addErrback(log.err)
d.addBoth(lambda _: reactor.stop())

# start the reactor to kick off connection estabilishing
reactor.run()
```

If you want you can use the *Cursor* class directly, with a interface closer to Psycopg. Note that using this method you have to make sure never to execute a query before the previous one finishes, as that would violate the PostgreSQL asynchronous protocol.

```
from txpostgres import txpostgres

from twisted.internet import reactor
from twisted.python import log, util

# define the libpq connection string and the query to use
connstr = 'dbname=postgres'
query = 'select tablename from pg_tables order by tablename'

# connect to the database
conn = txpostgres.Connection()
d = conn.connect('dbname=postgres')


def useCursor(cur):
    # execute a query
```

```
    d = cur.execute(query)
    # fetch the first row from the result
    d.addCallback(lambda _: cur.fetchone())
    # output it
    d.addCallback(lambda result: util.println('First table name:', result[0]))
    # and close the cursor
    return d.addCallback(lambda _: cur.close())

# create a cursor and use it
d.addCallback(lambda _: conn.cursor())
d.addCallback(useCursor)

# log any errors and stop the reactor
d.addErrback(log.err)
d.addBoth(lambda _: reactor.stop())

# start the reactor to kick off connection estabilishing
reactor.run()
```

## 1.1 Using transactions

Every query executed by txpostgres is committed immediately. If you need to execute a series of queries in a transaction, use the *runInteraction()* method:

```
from txpostgres import txpostgres

from twisted.internet import reactor
from twisted.python import log

# connect to the database
conn = txpostgres.Connection()
d = conn.connect('dbname=postgres')


def interaction(cur):
    """
    A callable that will execute inside a transaction.
    """
    # the parameter is a txpostgres Cursor
    d = cur.execute('create table test(x integer)')
    d.addCallback(lambda _: cur.execute('insert into test values (%s)', (1, )))
    return d

# run the interaction, making sure that if the insert fails, the table won't be
# left behind created but empty
d.addCallback(lambda _: conn.runInteraction(interaction))

# close the connection, log any errors and stop the reactor
d.addCallback(lambda _: conn.close())
d.addErrback(log.err)
d.addBoth(lambda _: reactor.stop())

# start the reactor to kick off connection estabilishing
reactor.run()
```

## 1.2 Customising the connection and cursor factories

You might want to customise the way txpostgres creates connections and cursors to take advantage of Psycopg features like dictionary cursors. To do that, define a subclass of *Connection* and override `connectionFactory` or `cursorFactory` class attributes to use your custom code. Here's an example of how to use dict cursors:

```
import psycopg2
import psycopg2.extras
from txpostgres import txpostgres

from twisted.internet import reactor
from twisted.python import log, util


def dict_connect(*args, **kwargs):
    kwargs['connection_factory'] = psycopg2.extras.DictConnection
    return psycopg2.connect(*args, **kwargs)


class DictConnection(txpostgres.Connection):
    connectionFactory = staticmethod(dict_connect)


# connect using the custom connection class
conn = DictConnection()
d = conn.connect('dbname=postgres')

# run a query and print the result
d.addCallback(lambda _: conn.runQuery('select * from pg_tablespace'))
# access the column by its name
d.addCallback(lambda result: util.println('All tablespace names:',
                                          [row['spcname'] for row in result]))

# close the connection, log any errors and stop the reactor
d.addCallback(lambda _: conn.close())
d.addErrback(log.err)
d.addBoth(lambda _: reactor.stop())

# start the reactor to kick off connection estabilishing
reactor.run()
```

## 1.3 Listening for database notifications

Being an asynchronous driver, txpostgres supports the PostgreSQL NOTIFY feature for sending asynchronous notifications to connections. Here is an example script that connects to the database and listens for notifications on the *list* channel. Every time a notification is received, it interprets the payload as part of the name of a table and outputs a list of tables with names containing that payload.

```
from txpostgres import txpostgres

from twisted.internet import reactor
from twisted.python import util


def outputResults(results, payload):
```

```
    print "Tables with `%s' in their name:" % payload
    for result in results:
        print result[0]


def observer(notify):
    if not notify.payload:
        print "No payload"
        return

    query = ("select tablename from pg_tables "
             "where tablename like '%%' || %s || '%%'")
    d = conn.runQuery(query, (notify.payload, ))
    d.addCallback(outputResults, notify.payload)


# connect to the database
conn = txpostgres.Connection()
d = conn.connect('dbname=postgres')

# add a NOTIFY observer
conn.addNotifyObserver(observer)
# start listening for NOTIFY events on the 'list' channel
d.addCallback(lambda _: conn.runOperation("listen list"))
d.addCallback(lambda _: util.println("Listening on the `list' channel"))

# process events until killed
reactor.run()
```

To try it execute the example code and then open another session using psql and try sending some NOTIFY events:

```
$ psql postgres
psql (9.1.2)
Type "help" for help.

postgres=> notify list, 'user';
NOTIFY
postgres=> notify list, 'auth';
NOTIFY
```

You should see the example program outputting lists of table names containing the payload:

```
$ python notify_example.py
Listening on the `list' channel
Tables with `user' in their name:
pg_user_mapping
Tables with `auth' in their name:
pg_authid
pg_auth_members
```

## 1.4 Automatic reconnection

The module includes provision for automatically reconnecting to the database in case the connection gets broken. To use it, pass a *DeadConnectionDetector* instance to *Connection*. You can customise the detector instance or subclass it to add custom logic. See the documentation for *DeadConnectionDetector* for details.

When a *Connection* is configured with a detector, it will automatically start the reconnection process whenever it encounters a certain class of errors indicative of a disconnect. See *defaultDeathChecker()* for more.

While the connection is down, all attempts to use it will result in immediate failures with *ConnectionDead*. This is to prevent sending additional queries down a link that's known to be down.

Here's an example of using automatic reconnection in txpostgres:

```
from txpostgres import txpostgres, reconnection

from twisted.internet import reactor, task


class LoggingDetector(reconnection.DeadConnectionDetector):

    def startReconnecting(self, f):
        print '[*] database connection is down (error: %r)' % f.value
        return reconnection.DeadConnectionDetector.startReconnecting(self, f)

    def reconnect(self):
        print '[*] reconnecting...'
        return reconnection.DeadConnectionDetector.reconnect(self)

    def connectionRecovered(self):
        print '[*] connection recovered'
        return reconnection.DeadConnectionDetector.connectionRecovered(self)


def result(res):
    print '-> query returned result: %s' % res


def error(f):
    print '-> query failed with %r' % f.value


def connectionError(f):
    print '-> connecting failed with %r' % f.value


def runLoopingQuery(conn):
    d = conn.runQuery('select 1')
    d.addCallbacks(result, error)


def connected(_, conn):
    print '-> connected, running a query periodically'
    lc = task.LoopingCall(runLoopingQuery, conn)
    return lc.start(2)


# connect to the database using reconnection
conn = txpostgres.Connection(detector=LoggingDetector())
d = conn.connect('dbname=postgres')

# if the connection failed, log the error and start reconnecting
d.addErrback(conn.detector.checkForDeadConnection)
d.addErrback(connectionError)
d.addCallback(connected, conn)
```

```
# process events until killed
reactor.run()
```

You can run this snippet and then try restarting the database. Logging lines should appear, as the connection gets automatically recovered.

## 1.5 Choosing a Psycopg implementation

To use txpostgres, you will need a recent enough version of Psycopg, namely 2.2.0 or later. Since parts of Psycopg are written in C, it is not available on some Python implementations, like PyPy. When first imported, txpostgres will try to detect if an API-compatible implementation of Psycopg is available.

You can force a certain implementation to be used by exporing an environment variable *TXPOSTGRES_PSYCOPG_IMPL*. Recognized values are:

**psycopg2** Force using Psycopg, do not try any fallbacks.

**psycopg2cffi** Use psycopg2cffi, a psycopg2 implementation based on cffi, known to work on PyPy.

**psycopg2ct** Use psycopg2ct, an older psycopg2 implementation using ctypes, also compatible with PyPy.

CHAPTER 2

# API documentation

All txpostgres APIs are documented here.

## 2.1 txpostgres.txpostgres

**class** `txpostgres.txpostgres.`**`Connection`**(*reactor=None*, *cooperator=None*, *detector=None*)
Bases: *`txpostgres.txpostgres._PollingMixin`*

A wrapper for a psycopg2 asynchronous connection.

The wrapper forwards almost everything to the wrapped connection, but provides additional methods for compatibility with adbapi.Connection.

**Parameters**

- **`reactor`** – A Twisted reactor or `None`, which means the current reactor
- **`cooperator`** – A Twisted Cooperator to process NOTIFY events or `None`, which means using task.cooperate

**Variables**

- **`connectionFactory`** (*any callable*) – The factory used to produce connections, defaults to psycopg2.connect
- **`cursorFactory`** (a callable accepting two positional arguments, a psycopg2.cursor and a *`Connection`*) – The factory used to produce cursors, defaults to *`Cursor`*

**`connect`**(**args*, ***kwargs*)
Connect to the database.

Any arguments will be passed to `connectionFactory`. Use them to pass database names, usernames, passwords, etc.

**Returns** A Deferred that will fire when the connection is open.

**Raise** *`AlreadyConnected`* when the connection has already been opened.

**`close`**()
Close the connection and disconnect from the database.

**Returns** `None`

**`cursor`**()
Create an asynchronous cursor using `cursorFactory`.

**runQuery**(**args*, ***kwargs*)
Execute an SQL query and return the result.

An asynchronous cursor will be created and its `execute()` method will be invoked with the provided arguments. After the query completes the results will be fetched and the returned Deferred will fire with the result.

The connection is always in autocommit mode, so the query will be run in a one-off transaction. In case of errors a Failure will be returned.

It is safe to call this method multiple times without waiting for the first query to complete.

**Returns** A Deferred that will fire with the return value of the cursor's `fetchall()` method.

**runOperation**(**args*, ***kwargs*)
Execute an SQL query and discard the result.

Identical to `runQuery()`, but the result won't be fetched and instead `None` will be returned. It is intended for statements that do not normally return values, like INSERT or DELETE.

It is safe to call this method multiple times without waiting for the first query to complete.

**Returns** A Deferred that will fire `None`.

**runInteraction**(*interaction*, **args*, ***kwargs*)
Run commands in a transaction and return the result.

`interaction` should be a callable that will be passed a `Cursor` object. Before calling `interaction` a new transaction will be started, so the callable can assume to be running all its commands in a transaction. If `interaction` returns a Deferred processing will wait for it to fire before proceeding. You should not close the provided `Cursor`.

After `interaction` finishes work the transaction will be automatically committed. If it raises an exception or returns a Failure the connection will be rolled back instead.

If committing the transaction fails it will be rolled back instead and the failure obtained trying to commit will be returned.

If rolling back the transaction fails the failure obtained from the rollback attempt will be logged and a `RollbackFailed` failure will be returned. The returned failure will contain references to the original failure that caused the transaction to be rolled back and to the `Connection` in which that happened, so the user can take a decision whether she still wants to be using it or just close it, because an open transaction might have been left open in the database.

It is safe to call this method multiple times without waiting for the first query to complete.

**Parameters** **interaction** (*any callable*) – A callable whose first argument is a `Cursor`.

**Returns** A Deferred that will fire with the return value of `interaction`.

**cancel**(*d*)
Cancel the current operation. The cancellation does not happen immediately, because the PostgreSQL protocol requires that the application waits for confirmation after the query has been cancelled. Be careful when cancelling an interaction, because if the interaction includes sending multiple queries to the database server, you can't really be sure which one are you cancelling.

**Parameters** **d** – a Deferred returned by one of `Connection` methods.

**cursorRunning**(*cursor*)
Called automatically when a `Cursor` created by this `Connection` starts polling after executing a query. User code should never have to call this method.

**cursorFinished**(*cursor*)
Called automatically when a *Cursor* created by this *Connection* is done with polling after executing a query. User code should never have to call this method.

**checkForNotifies**()
Check if NOTIFY events have been received and if so, dispatch them to the registered observers, using the Cooperator provided in the constructor. This is done automatically, user code should never need to call this method.

**addNotifyObserver**(*observer*)
Add an observer function that will get called whenever a NOTIFY event is delivered to this connection. Any number of observers can be added to a connection. Adding an observer that's already been added is ignored.

Observer functions are processed using the Cooperator provided in the constructor to avoid blocking the reactor thread when processing large numbers of events. If an observer returns a Deferred, processing waits until it fires or errbacks.

There are no guarantees as to the order in which observer functions are called when NOTIFY events are delivered. Exceptions in observers are logged and discarded.

**Parameters** **observer** (*any callable*) – A callable whose first argument is a psycopg2.extensions.Notify.

**removeNotifyObserver**(*observer*)
Remove a previously added observer function. Removing an observer that's never been added will be ignored.

**Parameters** **observer** (*any callable*) – A callable that should no longer be called on NOTIFY events.

**getNotifyObservers**()
Get the currently registered notify observers.

**Returns** A set of callables that will get called on NOTIFY events.

**Return type** set

**class** txpostgres.txpostgres.**Cursor**(*cursor*, *connection*)
Bases: *txpostgres.txpostgres._PollingMixin*

A wrapper for a psycopg2 asynchronous cursor.

The wrapper will forward almost everything to the wrapped cursor, so the usual DB-API interface can be used, but it will return Deferred objects that will fire with the DB-API results.

Remember that the PostgreSQL protocol does not support concurrent asynchronous queries execution, so you need to take care not to execute a query while another is still being processed.

In most cases you should just use the *Connection* methods that will handle the locking necessary to prevent concurrent query execution.

**execute**(*query*, *params=None*)
A regular DB-API execute, but returns a Deferred.

The caller must be careful not to call this method twice on cursors from the same connection without waiting for the previous execution to complete.

**Returns** A Deferred that will fire with the results of the DB-API execute.

**callproc**(*procname*, *params=None*)
A regular DB-API callproc, but returns a Deferred.

The caller must be careful not to call this method twice on cursors from the same connection without waiting for the previous execution to complete.

**Returns** A Deferred that will fire with the results of the DB-API callproc.

**close**()
Close the cursor.

Once closed, the cursor cannot be used again.

**Returns** None

**class** txpostgres.txpostgres.**ConnectionPool**(*_ignored*, **connargs*, ***connkw*)
Bases: object

A poor man's pool of *Connection* instances.

**Variables**

- **min** (*int*) – The amount of connections that will be open when start() is called. The pool never opens or closes connections on its own after starting. Defaults to 3.
- **connectionFactory** (*any callable*) – The factory used to produce connections, defaults to *Connection*.
- **reactor** – The reactor passed to connectionFactory.
- **cooperator** – The cooperator passed to connectionFactory.

**__init__**(*_ignored*, **connargs*, ***connkw*)
Create a new connection pool.

Any positional or keyword arguments other than the first one and a min keyword argument are passed to connectionFactory when connecting. Use these arguments to pass database names, usernames, passwords, etc.

**Parameters** **_ignored** (*any object*) – Ignored, for adbapi.ConnectionPool compatibility.

**start**()
Start the connection pool.

This will create as many connections as the pool's min variable says.

**Returns** A Deferred that fires when all connection have succeeded.

**close**()
Stop the pool.

Disconnects all connections.

**Returns** None

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

Provided to be able to remove broken connections from the pool. The caller should make sure the removed connection does not have queries pending.

**Parameters** **connection** (an object produced by the pool's connectionFactory) – The connection to be removed.

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

Provided to be able to extend the pool with new connections.

**Parameters** **connection** (an object compatible with those produced by the pool's `connectionFactory`) – The connection to be added.

**runQuery**(**args*, ***kwargs*)
Execute an SQL query using a pooled connection and return the result.

One of the pooled connections will be chosen, its *runQuery()* method will be called and the resulting Deferred will be returned.

**Returns** A Deferred obtained by a pooled connection's *runQuery()*

**runOperation**(**args*, ***kwargs*)
Execute an SQL query using a pooled connection and discard the result.

One of the pooled connections will be chosen, its *runOperation()* method will be called and the resulting Deferred will be returned.

**Returns** A Deferred obtained by a pooled connection's *runOperation()*

**runInteraction**(*interaction*, **args*, ***kwargs*)
Run commands in a transaction using a pooled connection and return the result.

One of the pooled connections will be chosen, its *runInteraction()* method will be called and the resulting Deferred will be returned.

**Parameters** **interaction** (*any callable*) – A callable that will be passed to `Connection.runInteraction`

**Returns** A Deferred obtained by a pooled connection's `Connection.runInteraction`

**class** `txpostgres.txpostgres.`**_PollingMixin**
Bases: `object`

An object that wraps something pollable. It can take care of waiting for the wrapped pollable to reach the OK state and adapts the pollable's interface to interfaces.IReadWriteDescriptor. It will forward all attribute access that is has not been wrapped to the underlying pollable. Useful as a mixin for classes that wrap a psycopg2 pollable object.

**Variables**

- **reactor** (an IReactorFDSet provider) – The reactor that the class will use to wait for the wrapped pollable to reach the OK state.
- **prefix** (`str`) – Prefix used during log formatting to indicate context.

**pollable**()
Return the pollable object. Subclasses should override this.

**Returns** A psycopg2 pollable.

**poll**()
Start polling the wrapped pollable.

**Returns** A Deferred that will fire with an instance of this class when the pollable reaches the OK state.

**continuePolling**(*swallowErrors=False*)
Move forward in the poll cycle. This will call psycopg2's poll() on the wrapped pollable and either wait for more I/O or callback or errback the Deferred returned earlier if the polling cycle has been completed.

**Parameters** **swallowErrors** (*bool*) – Should errors with no one to report them to be ignored.

**Raise** *UnexpectedPollResult* when `poll()` returns a result from outside of the expected list.

**exception** txpostgres.txpostgres.**AlreadyConnected**
Bases: exceptions.Exception

The database connection is already open.

**exception** txpostgres.txpostgres.**RollbackFailed**(*connection*, *originalFailure*)
Bases: exceptions.Exception

Rolling back the transaction failed, the connection might be in an unusable state.

**Variables**

- **connection** (*Connection*) – The connection that failed to roll back its transaction.
- **originalFailure** (a Twisted Failure) – The failure that caused the connection to try to roll back the transaction.

**exception** txpostgres.txpostgres.**UnexpectedPollResult**
Bases: exceptions.Exception

Polling returned an unexpected result.

**exception** txpostgres.txpostgres.**AlreadyPolling**
Bases: exceptions.Exception

The previous poll cycle has not been finished yet.

This probably indicates an issue in txpostgres, rather than in user code.

## 2.2 txpostgres.reconnection

**class** txpostgres.reconnection.**DeadConnectionDetector**(*deathChecker=None*, *reconnectionIterator=None*, *reactor=None*)
Bases: object

A class implementing reconnection strategy. When the connection is discovered to be dead, it will start the reconnection process.

The object being reconnected should proxy all operations through the detector's *callChecking()* which will automatically fail them if the connection is currently dead. This is done to prevent sending requests to a resource that's not currently available.

When an instance of *Connection* is passed a *DeadConnectionDetector* it automatically starts using it to provide reconnection.

Another way of using this class is manually calling *checkForDeadConnection()* passing it a Failure instance to trigger reconnection. This is useful to handle initial connection errors, for example:

```
conn = txpostgres.Connection(detector=DeadConnectionDetector())
d = conn.connect('dbname=test')
d.addErrback(conn.detector.checkForDeadConnection)
```

**Variables**

- **reconnectable** (object) – An object to be reconnected. It should provide a *connect* and a *close* method.
- **connectionIsDead** (bool) – If the connection is currently believed to be dead.

**setReconnectable**(*reconnectable*, **connargs*, ***connkw*)
Register a reconnectable with the detector. Needs to be called before the detector will be used. The remaining arguments will be passed to the reconnectable's *connect* method on each reconnection.

**Parameters** **reconnectable** (object) – An object to be reconnected. It should provide a *connect* and a *close* method.

**callChecking**(*method*, **args*, ***kwargs*)
Call a method if the connection is still alive.

**checkForDeadConnection**(*f*)
Get passed a Failure instance and determine if it means that the connection is dead. If so, start reconnecting.

**startReconnecting**(*f*)
Called when the connection is detected to be dead.

**reconnect**()
Called on each attempt of reconnection.

**connectionRecovered**()
Called when the connection has recovered.

**addRecoveryHandler**(*handler*)
Add a handler function that will get called whenever the connection is recovered. Any number of handlers can be added. Adding a handler that's already been added is ignored.

Recovery handlers are ran in parallel. If any of them return a Deferred, recovery will wait until it fires.

There are no guarantees as to the order in which handler functions are called. Exceptions in handlers are logged and discarded.

**Parameters** **handler** – A zero-argument callable.

**removeRecoveryHandler**(*handler*)
Remove a previously added recovery handler. Removing a handler that's never been added will be ignored.

**Parameters** **handler** – A callable that should no longer be called when the connection recovers.

**getRecoveryHandlers**()
Get the currently registered recovery handlers.

**Returns** A set of callables that will get called on recovery.

**Return type** set

txpostgres.reconnection.**defaultDeathChecker**(*f*)
Checker function suitable for use with *DeadConnectionDetector*.

txpostgres.reconnection.**defaultReconnectionIterator**()
A function returning sane defaults for a reconnection iterator, for use with *DeadConnectionDetector*.

The defaults have maximum reconnection delay capped at 10 seconds and no limit on the number of retries.

**exception** txpostgres.reconnection.**ConnectionDead**
Bases: exceptions.Exception

The connection is dead.

## 2.3 txpostgres.retrying

**class** `txpostgres.retrying.`**`RetryingCall`**(*f*, **args*, ***kw*)

Bases: `object`

Calls a function repeatedly, passing it args and keyword args. Failures are passed to a user-supplied failure testing function. If the failure is ignored, the function is called again after a delay whose duration is obtained from a user-supplied iterator. The start method (below) returns a Deferred that fires with the eventual non-error result of calling the supplied function, or fires its errback if no successful result can be obtained before the delay backoff iterator raises `StopIteration`.

It is important to note the behaviour when the delay of any of the steps is zero. The function is the called synchronously, ie. control does not go back to the reactor between obtaining the delay from the iterator and calling the function if the iterator returns zero.

The *`resetBackoff()`* method replaces the backoff iterator with another one and is useful to reset the delay if some phase of the process has succeeded and that makes the desirable initial delay different again.

**`start`**(*backoffIterator=None*, *failureTester=None*)

Start the call and retry it until it succeeds and fails.

**Parameters**

- **`backoffIterator`** (*callable*) – A zero-argument callable that should return a iterator yielding reconnection delay periods. If `None` then *`simpleBackoffIterator()`* will be used.
- **`failureTester`** (*callable*) – A one-argument callable that will be called with a Failure instance each time the function being retried fails. It should return `None` if the call should be retried or a Failure if the retrying process should be stopped. If `None` is used for this parameter, retrying will never stop until the backoff iterator is exhausted.

**`resetBackoff`**(*backoffIterator=None*)

Replace the current backoff iterator with a new one.

`txpostgres.retrying.`**`simpleBackoffIterator`**(*initialDelay=1.0*, *maxDelay=3600*, *factor=2.718281828459045*, *jitter=0.11962656472*, *maxRetries=10*, *now=True*)

Yields increasing timeout values between retries of a call. The default factor and jitter are taken from Twisted's ReconnectingClientFactory.

**Variables**

- **`initialDelay`** (`float`) – Initial delay, in seconds.
- **`maxDelay`** (`float`) – Maximum cap for the delay, if zero then no maximum is applied.
- **`factor`** (`float`) – Multiplicative factor for increasing the delay.
- **`jitter`** (`float`) – Randomness factor to include when increasing the delay, to prevent stampeding.
- **`maxRetries`** (`int`) – If non-zero, only yield so many values after exhausting the iterator.
- **`now`** (`bool`) – If the very first delay yielded should always be zero.

CHAPTER 3

# Indices and tables

- genindex
- search

## t

## Symbols



## A



## C



## D



## E



## G



## P



## R

## S



## T



## U