Rdbhdb Documentation

Release 0.9.6

David Keeney

May 02, 2014

The Rdbhost.com service provides SQL databases accessible with the full power of SQL, directly from the browser, or from your remote Python app.

The **Rdbhdb** module provides a Python DB API v2 interface for working with SQL databases hosted on Rdbhost.com. As of version 0.9.6, it also supports asynchronous access under the Asyncio framework.

For this module to be at all useful, you need an account on Rdbhost.com; accounts are free and easy to create, at the http://www.rdbhost.com website. Once logged into your account, visit the 'Role Manager' page and cut and paste the super *role name* and *authcode*.

The module works with Python 2.5 and up, or Python 3.1 and up. The Asyncio features are only available for version 3.4 (and I assume 3.3 with Asyncio installed).

Installation:

```
pip install rdbhdb
```

Import the module:

```
from rdbhdb import rdbhdb
from rdbhdb import extensions
```

The *extensions* submodule is optional, and provides dictionary style cursors.

Open a connection with the `connect` function. If you are using Asyncio, add a keyword parameter `asyncio=True`.

```
conn = rdbhdb.connect(<role>, <authcode>)
conn = rdbhdb.connect(<role>, <authcode>, asyncio=True)
```

Create a cursor from the connection. If you want a dictionary cursor, provide a cursor factory, and if you are using Asyncio, use an async cursor factory:

```
cur = conn.cursor()
dictCur = conn.cursor(cursor_factory=extensions.DictCursor)
asyncCur = conn.cursor(cursor_factory=extensions.AsyncDictCursor)
```

To submit a query to the server, use one of the `.execute` methods:

```
cur.execute(<sql>, <params>)
cur.executemany(<sql>, <set of params>)
```

The *sql* can be one SQL statement, or an aggregate of multiple statements joined with ';'. Params can be a list or a dictionary, depending on value of .paramstyle. For *.executemany()*, params is a list of param sets, each of which is a dictionary or list.

With Asyncio, these methods are coroutines, and call them with `yield from`:

```
yield from cur.execute(<sql>, <params>)
```

You can, alternatively, use callbacks by creating a Future and attaching the callback:

```
task = asyncio.Task(cur.execute(<sql>, <params<))
task.add_done_callback(when_done)

yield from async.wait([task])
```

If the query will yield more than 100 records, and you do not provide an explicit *LIMIT* clause in the query, the *execute* call will raise an exception. With an explicit *LIMIT* clause, you can retrieve up to 1000 records.

After the execute method returns, the data is available on the cursor using the *.fetch* methods:

```
row = cur.fetchone()
rows = cur.fetchmany(<ct>)
rows = cur.fetchall()
```

Each of these can be called repeatedly, until all data is received. These are plain methods, even under Asyncio, as the data is on the client already when the *.execute* completes, and there is no further fetching wait.

If the *sql* was multiple statements, then each will have its own set of results, and `.nextset` will advance to the next. Each statement will have a result set, possibly empty.

```
cur.execute('SELECT 1 AS one; SELECT 2 AS two;',)
first = cur.fetchall()     # ({'one': 1}, )
cur.nextset()              # True
second = cur.fetchall()    # ({'two': 2}, )
cur.nextset()              # None
```

String data provided as parameters can be *unicode* or *'utf-8'* encoded strings. Data that is to be handled as binary (no decoding) should be `bytes` type in Python 3 or `buffer` in Python 2.

Some attributes of the cursor, required by the API, are:

```
.description = tuple of tuples, one tuple for each field/column
    each field-tuple is (name, type_code, None, None, None, None, None)
    type_code is the PostgreSQL field type code for the field.
.rowcount = how many rows were retrieved by cursor.
.array_size = default count for .fetchmany()
```

Some additional methods of the cursor are:

```
.close() - closes cursor. further use raises exception.
```

The connection objects have the following methods:

```
.close() - closes connection. further use raises exception.
.commit() - a noop, as each request is automatically wrapped in a transaction, with closing commit.
```

The API defines this, but has no meaning for the Rdbhost implementaion.:

```
.rollback() - not implemented, so use will raise exception.
```