dbConnect Documentation

Release 1.6.0

Emin Mastizada

Apr 02, 2017

Contents

Release v1.6.0. (*Installation*)

dbConnect is an *MPLv2 Licensed* Module for **little projects** using *mysql* or *postgresql* databases. It generates mysql and postgresql queries automatically, you just send data in pythonic style and it does the rest.

```
>>> from dbConnect import DBConnect
>>> database = DBConnect('credentials.json')
>>> users = database.fetch('users', limit=5, filters={'status': 'active'})
>>> new_user = {'name': 'Emin', 'status': 'active', 'company': 'pyninjas'}
>>> database.insert(new_user, 'users')
```

CHAPTER 1

Feature Support

- **fetch** all fields in table as dictionary (column name: value)
- fetch only selected fields
- fetch using filters
- limit fetch result
- filter case [AND, OR]
- **insert** to table
- **update** row
- **delete** row
- **increment** column in table
- **sum** of a numeric column(s)
- **custom sql query**

CHAPTER 2

User Guide

Introduction

dbConnect vs ORM

If you have big project then building models (entities) and using ORM will help you a lot and will be a lot safer.

dbConnect was made as little module to be used in small projects that need to do some interactions with MySQL or PostgreSQL databases.

It's just a big time saver for developers and helps to keep your code clean and readable.

dbConnect License

dbConnect is released under terms of MPLv2 License.

Installation

This part of the documentation covers the installation of dbConnect.

Requirements

dbConnect uses mysql.connector for mysql, install it using:

```
$ apt-get install python3-mysql.connector
$ apt-get install python-mysql.connector
```

For PostgreSQL install psycopg2 module:

```
$ pip install psycopg2
```

Distribute & Pip

Installing dbConnect is simple with pip, just run this in your terminal:

```
$ pip install dbConnect
```

or, with easy_install:

```
$ easy_install dbConnect
```

But, you really shouldn't do that.

Get the Code

dbConnect is actively developed on GitHub, where the code is always available.

You can either clone the public repository:

```
$ git clone git@github.com:mastizada/dbConnect.git
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily:

```
$ python setup.py install
```

Quickstart

class `dbConnect.DBConnect`

This page gives a good introduction in how to get started with dbConnect.

First, make sure that:

- dbConnect is *installed*

Lets go over every function.

Connection

dbConnect uses `credentials.json` file from your project directory by default. You can give custom path for that file:

```
>>> database = DBConnect('/home/user/project/credentials.json')
```

Or provide database details:

```
>>> database = DBConnect(host='127.0.0.1', user='root', password='', database='test')
```

You can provide any other parameters that are available in mysql.connector or PostgreSQL Documentation

- After successfull connection there will be **database.connection** and

database.cursor variables that can be used as in official MySQL or PostgreSQL documentation.

Engines

dbConnect supports *mysql* and *postgres* as *engine* options.

Fetch Data

Get data from table.

Basic usage:

```
>>> database.fetch('table_name')
```

Fields:

- table: `str` : name of table, must be provided
- limit: `int` : result limit, default `1000`
- fields: `array` : list of column names, default `None`
- filters: `dict` : dictionary with keys as column name, default `None`
- case: `str` : search case for filter [AND, OR], default `'AND'`

Example:

```
>>> database.fetch('user', limit=5, fields=['id', 'name', 'email'], filters={'company
↪': 'pyninjas'})
>>> database.fetch('user', limit=5, filters={'id': (10, '>=')})  # Get 5 user whose␣
↪id is higher than 10
>>> database.fetch('user', filters={'email': (None, 'is')}) # Get users whose email␣
↪is NULL
>>> database.fetch('user', filters={'email': None}) # Same as (None, 'is')
>>> database.fetch('user', filters={'email': (None, 'is not')}) # Get users whole␣
↪email is not NULL
>>> database.fetch('user', filters={'id': (0, 100, '<=>')}) # Get users whose id is␣
↪between 0 and 100
>>> database.fetch('user', filters={'id': (0, 100, '<>')}) # Get users whose id is␣
↪between 1 and 99
```

Insert Data

Add new row (data) to table.

Fields:

- data: `dict` : dictionary with keys as column name, must be provided
- table: `str` : name of table, must be provided
- commit: `bool` : commit after insert command, default: True
- update: `dict` : Update selected columns if key is duplicate, default: None

Example:

```
>>> new_user = {'name': 'Emin', 'company': 'pyninjas', 'website': 'mastizada.com'}
>>> database.insert(new_user, 'user')  # Adds new_user to user table
```

Example 2:

```
>>> new_user = {'id': 1, 'name': 'Ramin', 'company': 'pyninjas', 'website':
↪'mastizada.com'}
>>> # if there is user with id=1, then update its name:
>>> updated_columns = {'name': 'Ramin'}
>>> database.insert(new_user, 'user', update=updated_columns)
```

Update Data

Update existing row.

Fields:

- data: `dict` : dictionary with keys as column name that will be changed, must be provided
- filters: `dict` : filters to find row(s) that will be changed, must be provided
- table: `str` : name of table, must be provided
- case: `str` : search case for filter [AND, OR], default `'AND'`
- commit: `bool` : commit after insert command, default: True

Example:

```
>>> database.update({'name': 'Emin Mastizada'}, {'id': 1, 'name': 'Emin'}, 'user',␣
↪case='OR')
```

Delete Data

Delete row from database.

Fields:

- table: `str` : name of table, must be provided
- filters: `dict` : filters to find row(s) that will be deleted, must be provided
- case: `str` : search case for filter [AND, OR], default `'AND'`
- commit: `bool` : commit after insert command, default: True

Example:

```
>>> database.delete('user', {'id': 1, 'name': 'Emin Mastizada'}, case='OR')
```

Increment Columns

Increment provided columns.

Fields:

- table: `str` : name of table, must be provided
- fields: `array` : list of column names to increment, required

- steps: `int` : Steps to increment, must be provided
- filters: `dict` : filters to find row(s)
- case: `str` : search case for filter [AND, OR], default `'AND'`
- commit: `bool` : commit after insert command, default: True

Example:

```
>>> database.increment('user', ['views'], steps=2, filters={'id': 1})
```

Total sum of a numeric column(s).

Fields:

- table: `str` : name of table, must be provided
- fields: `array` : list of numeric column names, required
- filters: `dict` : filters to find row(s)
- case: `str` : search case for filter [AND, OR], default `'AND'`

Example:

```
>>> database.value_sum('user', fields=['views'])
```

Custom SQL Query

Execute custom sql queries when you need something complex.

Example:

```
>>> database.cursor.execute("SELECT * FROM table WHERE id = 5")
>>> results = database.cursor.fetchall()
```

```
>>> database.cursor.execute("SELECT u.name FROM users as u INNER JOIN tasks as t ON t.
↪user = u.id WHERE t.progress = 'assigned'")
>>> users = database.cursor.fetchall()
```

Commit Data

Commit changes to database.

No fields.

Example:

```
>>> database.commit()
```

And now enjoy and give me your feedbacks ;)

Python Module Index

d

Index

D