Privex Python Database Wrappers Documentation

Privex Inc., Chris (Someguy123)

Feb 24, 2020

MAIN:

Welcome to the documentation for Privex's `Python Database Wrappers`_ - lightweight classes and functions to ease managing and interacting with various relation database systems (RDBMS's), including SQLite3 and PostgreSQL.

This documentation is automatically kept up to date by ReadTheDocs, as it is automatically re-built each time a new commit is pushed to the Github Project

Contents

CHAPTER

ONE

QUICK INSTALL

Installing with Pipenv (recommended)

```
pipenv install privex-db
```

Installing with standard `pip3`

```
pip3 install privex-db
```

CHAPTER

TWO

ALL DOCUMENTATION

2.1 Installing Privex Python DB Wrappers

2.1.1 Download and install from PyPi using pipenv / pip (recommended)

Installing with Pipenv **(recommended)**

```
pipenv install privex-db
```

Installing with standard `pip3`

```
pip3 install privex-db
```

2.1.2 (Alternative) Manual install from Git

You may wish to use the alternative installation methods if:

- You need a feature / fix from the Git repo which hasn't yet released as a versioned PyPi package
- You need to install privex-db on a system which has no network connection
- You don't trust / can't access PyPi
- For some reason you can't use `pip` or `pipenv`

Option 1 - Use pip to install straight from Github

```
pip3 install git+https://github.com/Privex/python-db
```

Option 2 - Clone and install manually

```
# Clone the repository from Github
git clone https://github.com/Privex/python-db
cd python-db

# RECOMMENDED MANUAL INSTALL METHOD
# Use pip to install the source code
pip3 install .

# ALTERNATIVE MANUAL INSTALL METHOD
# If you don't have pip, or have issues with installing using it, then you can use␣
↪setuptools instead.
python3 setup.py install
```

2.2 Examples / Basic Usage

2.2.1 Using the SQLite3 Manager + Query Builder

2.2.1.1 Basic / direct usage of SqliteWrapper

```
from os.path import expanduser
from typing import List, Tuple
from privex.db import SqliteWrapper

# Open or create the database file ~/.my_app/my_app.db
db = SqliteWrapper(expanduser("~/.my_app/my_app.db"))

# Create the table 'items' and insert some items
db.action("CREATE TABLE items (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT);")
db.action("INSERT INTO items (name) VALUES (?);", ["Cardboard Box"])
db.action("INSERT INTO items (name) VALUES (?);", ["Orange"])
db.action("INSERT INTO items (name) VALUES (?);", ["Banana"])
db.action("INSERT INTO items (name) VALUES (?);", ["Stack of Paper"])

item = db.fetchone("SELECT * FROM items WHERE name = ?", ['Orange'])

print(item.id, '-', item.name)
# Output: 2 - Orange
```

2.2.1.2 Using the query builder (SqliteQueryBuilder)

Once you have an instance of *SqliteWrapper*, you can easily create query builders via the `.builder` function.

Privex-DB query builders work similarly to Django's ORM, and are very simple to use.

```
q = db.builder('items')
# Privex QueryBuilder's support chaining similar to Django's ORM
q.select('id', 'name') \            # SELECT id, name
    .where('name', 'Orange') \      # WHERE name = 'Orange'
    .where_or('name', 'Banana') \   # OR name = 'Banana'
    .order('name', 'id')            # ORDER BY name, id DESC

# You can either iterate directly over the query builder object
for row in q:
    print(f"ID: {row.id}   Name: {row.name}")
# Output:
# ID: 3   Name: Banana
# ID: 2   Name: Orange

# Or you can use .fetch / .all to grab a single row, or all rows as a list
item = db.builder('items').where('name', 'Orange').fetch()
# {'id': 2, 'name': 'Orange'}
items = db.builder('items').all()
# [ {'id': 1, 'name': 'Cardboard Box'}, {'id': 2, 'name': 'Orange'}, ... ]
```

2.2.1.3 Sub-classing SqliteWrapper for your app

To make the most out of the wrapper classes, you'll want to create a sub-class which is tuned for your application, including the table schemas that your application needs.

NOTE: *SqliteWrapper* runs in auto-commit mode by default. If you don't want to use auto-commit, you can pass `isolation_level=XXX` to the constructor to choose a custom isolation level without autocommit. See the Python SQLite3 Docs for more information on isolation modes.

Below is an example of sub-classing *SqliteWrapper* to create two tables (`users` and `items`), with some custom helper methods, then instantiating the class, inserting some rows, and querying them.

```
from os.path import expanduser, join
from typing import List, Tuple
from privex.db import SqliteWrapper

class MyDBWrapper(SqliteWrapper):
    ###
    # If a database path isn't specified, then the class attribute DEFAULT_DB will be␣
↪used.
    ###
    DEFAULT_DB_FOLDER: str = expanduser('~/.my_app')
    DEFAULT_DB_NAME: str = 'my_app.db'
    DEFAULT_DB: str = join(DEFAULT_DB_FOLDER, DEFAULT_DB_NAME)

    ###
    # The SCHEMAS class attribute contains a list of tuples, with each tuple␣
↪containing the name of a
    # table, as well as the SQL query required to create the table if it doesn't␣
↪exist.
    ###
    SCHEMAS: List[Tuple[str, str]] = [
        ('users', "CREATE TABLE users ("
                  "id INTEGER PRIMARY KEY AUTOINCREMENT, "
                  "first_name TEXT, "
                  "last_name TEXT, "
                  "address TEXT NULL"
                  ");"),
        ('items', "CREATE TABLE items (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"),
    ]

    def get_items(self):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchall with a pre-defined SQL query
        return self.fetchall("SELECT * FROM items;")

    def find_item(self, id: int):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchone with a pre-defined SQL query, and interpolates the 'id'␣
↪parameter into
        # the prepared statement.
        return self.fetchone("SELECT * FROM items WHERE id = ?;", [id])

    def get_users(self): return self.fetchall("SELECT * FROM users;")
```

(continues on next page)

(continued from previous page)

```
    def find_user(self, id: int): return self.fetchall("SELECT * FROM users WHERE id␣
↪= ?;", [id])

# Once the class is constructed, it should've created the SQLite3 database ~/.my_app/
↪my_app.db (if it didn't exist)
# and then created the tables 'users' and 'items' if they didn't already exist.
db = MyDBWrapper()

# The method .action runs a query, but doesn't attempt to fetch rows, it only returns␣
↪the affected row count
# Note: By default, SqliteWrapper uses SQLite3 auto-commit mode
db.action("INSERT INTO users (first_name, last_name) VALUES (?, ?);", ['John', 'Doe'])
db.action("INSERT INTO users (first_name, last_name, address) VALUES (?, ?, ?);", [
↪'Jane', 'Doe', '123 Ex St'])
db.action("INSERT INTO users (first_name, last_name) VALUES (?, ?);", ['Dave',
↪'Johnston'])
db.action("INSERT INTO users (first_name, last_name) VALUES (?, ?);", ['Aaron',
↪'Johnston'])

users = db.get_users()

for u in users:
    print(f"User: ID {u.id}  /  First Name: {u.first_name}   /   Last Name: {u.last_
↪name}")
```

If we then run this example, we get the output:

```
user@example ~ $ python3 example.py
User: ID 1  /  First Name: John   /   Last Name: Doe
User: ID 2  /  First Name: Jane   /   Last Name: Doe
User: ID 3  /  First Name: Dave   /   Last Name: Johnston
User: ID 4  /  First Name: Aaron   /   Last Name: Johnston
```

2.2.1.4 Using the query builder from your sub-class

We can also use *SqliteQueryBuilder* directly from our sub-class, which is a primitive ORM for building and executing SQL queries.

Let's build a slightly complex query to show how powerful it is. We'll build a query to aggregate the number of users who share a given last name AND don't have an address.

```
# Get an SqliteQueryBuilder instance for the table 'users'
q = db.builder('users')

# Privex QueryBuilder's support chaining similar to Django's ORM
q \
    .select('last_name', 'COUNT(last_name) AS total') \
    .where('address', None) \
    .group_by('last_name')

print(f"\nQuery:\n\t{q.build_query()}\n")
results = q.all()

for r in results:
    print('Result:', r)
```

If we then run this example, we get the output:

```
Query:
     SELECT last_name, COUNT(last_name) AS total FROM users WHERE address IS NULL␣
↪GROUP BY last_name;

Result: {'last_name': 'Doe', 'total': 1}
Result: {'last_name': 'Johnston', 'total': 2}
```

privex.db.base	This module contains core functions/classes which are used across the module, as well as abstract classes / types.
privex.db.postgres	This module holds *PostgresWrapper* - a somewhat higher level class for interacting with PostgreSQL.
privex.db.sqlite	This module holds *SqliteWrapper* - a somewhat higher level class for interacting with SQLite3 databases.
privex.db.types	This module holds newly defined types which are used across the module, such as *GenericCursor* and *GenericConnection*
privex.db.query	

2.3 privex.db.base

This module contains core functions/classes which are used across the module, as well as abstract classes / types.

Copyright:

```
    +==================================================+
    |                 © 2019 Privex Inc.               |
    |               https://www.privex.io              |
    +==================================================+
    |                                                  |
    |        Privex's Python Database Library          |
    |        License: X11 / MIT                        |
    |                                                  |
    |        Originally Developed by Privex Inc.       |
    |        Core Developer(s):                        |
    |                                                  |
    |          (+)  Chris (@someguy123) [Privex]       |
    |                                                  |
    +==================================================+

Copyright (c) 2019     Privex Inc.   ( https://www.privex.io )
```

Classes

CursorManager(cursor[, close_callback])	Not all database API's support context management with their cursors, so this class wraps a given database cursor objects, and provides context management methods `__enter__()` and `__exit__()`
DBExecution(*args, **kwargs)	
GenericDBWrapper([db, connector_func])	This is a generic database wrapper class, which implements various methods such as:

2.3.1 CursorManager

class privex.db.base.**CursorManager**(*cursor: CUR*, *close_callback: Optional[callable] = None*)

Not all database API's support context management with their cursors, so this class wraps a given database cursor objects, and provides context management methods `__enter__()` and `__exit__()`

__init__(*cursor: CUR*, *close_callback: Optional[callable] = None*)

Initialise the cursor manager.

Parameters

- **cursor** (*CUR | GenericCursor*) – A database cursor object to wrap
- **close_callback** (*callable*) – If specified, this callable (function/method) will be called BEFORE and AFTER the cursor is closed, with the kwargs `state='BEFORE_CLOSE'` and `state='AFTER_CLOSE'` respectively.

can_cleanup: bool = None

This becomes True if this is the **first** context manager instance for a cursor

is_context_manager: bool = None

`True` if this class is being used in a `with` statement, otherwise `False`.

2.3.1.1 Methods

Methods

__init__(cursor[, close_callback])	Initialise the cursor manager.
close(*args, **kwargs)	
execute(*args, **kwargs)	
fetchall(*args, **kwargs)	
fetchmany(*args, **kwargs)	
fetchone(*args, **kwargs)	

2.3.1.1.1 __init__

CursorManager.**__init__**(*cursor: CUR*, *close_callback: Optional[callable] = None*)

Initialise the cursor manager.

Parameters

- **cursor** (*CUR | GenericCursor*) – A database cursor object to wrap
- **close_callback** (*callable*) – If specified, this callable (function/method) will be called BEFORE and AFTER the cursor is closed, with the kwargs `state='BEFORE_CLOSE'` and `state='AFTER_CLOSE'` respectively.

2.3.1.1.2 close

CursorManager.**close**(**args*, ***kwargs*)

2.3.1.1.3 execute

CursorManager.**execute**(**args*, ***kwargs*)

2.3.1.1.4 fetchall

CursorManager.**fetchall**(**args*, ***kwargs*)

2.3.1.1.5 fetchmany

CursorManager.**fetchmany**(**args*, ***kwargs*)

2.3.1.1.6 fetchone

CursorManager.**fetchone**(**args*, ***kwargs*)

2.3.1.2 Attributes

Attributes

description
lastrowid
rowcount

2.3.1.2.1 description

property CursorManager.**description**

2.3.1.2.2 lastrowid

property CursorManager.**lastrowid**

2.3.1.2.3 rowcount

property CursorManager.**rowcount**

2.3.2 DBExecution

class privex.db.base.**DBExecution**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)

2.3.2.1 Methods

Methods

—

2.3.2.2 Attributes

Attributes

—

2.3.3 GenericDBWrapper

class privex.db.base.**GenericDBWrapper**(*db=None*, *connector_func: callable = None*, ***kwargs*)

This is a generic database wrapper class, which implements various methods such as:

- Querying methods such as *query()*, *fetch()*, *fetchone()*, *fetchall()*
- Table management functions such as *create_schemas()*, *drop_schemas()* and *drop_table()*
- Connection and cursor methods / properties: *conn*, *get_cursor()*

While this class is intended to be subclassed by DBMS-specific wrapper classes, all methods follow the Python DB API (PEP 249) and the ANSI SQL standard, meaning very little modification is actually required to adapt this wrapper to most database systems.

See *PostgresWrapper* and *SqliteWrapper* for implementation examples.

__init__(*db=None*, *connector_func: callable = None*, ***kwargs*)
Initialise the database wrapper class.

This constructor sets `_conn` to None, and sets up various instance variables such as `connector_func` and `query_mode`.

While you can set various instance variables such as `query_mode` via this constructor, if you're inheriting this class, it's recommended that you override the `DEFAULT_` static class attributes to your preference.

Parameters

- **db** (*str*) – The name / path of the database to connect to
- **connector_func** (*callable*) – A function / method / lambda which returns a database connection object which acts like *GenericConnection*

Key bool auto_create_schema (Default: `True`) If True, call *create_schemas()* during constructor.

Key list connector_args A list of arguments to be passed as positional args to `connector_func`

Key dict connector_kwargs A dict of arguments to passed as keyword args to `connector_func`

Key str query_mode Either `flat` (return tuples) or `dict` (return dicts of column:value) Controls how results are returned from query functions, e.g. *query()* and *fetch()*

Key str table_query The query used to check for existence of a table. The query should take one prepared statement argument (the table name to check for), and the first column returned must be named `table_count` - an integer containing how many tables were found under the given name (usually just 0 if not found, 1 if found).

Key str table_list_query The query used to obtain a list of tables in the database. The query should take no arguments, and return rows containing one column each, `name` - the name of the table.

AUTO_ZIP_COLS: bool = True

If your database API doesn't support returning rows as dicts or a dict-like structure (e.g. sqlite3), then when this setting is enabled, `_zip_cols()` will be called to zip each row with the result column names into a dictionary.

If your database API supports returning rows as a dictionary either by default, or via a cursor/connection class (e.g. PostgreSQL with psycopg2) then you should set this to `False` and use the cursor/connection class instead.

DEFAULT_PLACEHOLDER: str = '?'

The placeholder used by the database API for prepared statements in .execute()

DEFAULT_QUERY_MODE: str = 'dict'

This attribute can be overridden on your inheriting wrapper class to change the default query mode used if one isn't specified in the constructor.

SCHEMAS: List[Tuple[str, str]] = []

This should be set as a class attribute to a list of two value tuples, each containing the name of a table, and the SQL query to create the table if it doesn't exist.

Example:

```
SCHEMAS = [
    (
        'my_table',
        "CREATE TABLE my_table (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"
    ),
```

(continues on next page)

(continued from previous page)

```
    (
        'other_table',
        "CREATE TABLE other_table (id INTEGER PRIMARY KEY AUTOINCREMENT,␣
↪example TEXT);"
    ),
]
```

action(*sql: str*, **params*, ***kwparams*) → int

Use *action()* as a simple alias method for running "action" queries which don't return results, only affected row counts.

For example `INSERT`, `UPDATE`, `CREATE` etc. queries.

This method calls *query()* with `fetch='no'`, saves the row count into a variable, closes the cursor, then returns the affected row count as an integer.

```
>>> db = GenericDBWrapper('SomeDB')
>>> rows_affected = db.action("DELETE FROM users WHERE first_name LIKE 'John%
↪';")
>>> rows_affected
4
```

Parameters

- **sql** (*str*) – An SQL query to execute on the current DB, as a string.
- **params** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`
- **kwparams** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`

Return int row_count Number of rows affected

property conn

Get or create a database connection

create_schema(*table: str*, *schema: str = None*)

Create the individual table `table`, either uses the create statement `schema`, or if `schema` is empty, then checks for a pre-existing CREATE statement for `table` in *SCHEMAS*.

```
>>> db = GenericDBWrapper('SomeDBName')
>>> db.create_schema('users', 'CREATE TABLE users (id INT PRIMARY KEY, name␣
↪VARCHAR(200));')
```

Parameters

- **table** –
- **schema** –

Returns

create_schemas(**tables*) → dict

Create all tables listed in *SCHEMAS* if they don't already exist.

Parameters **tables** (*str*) – If one or more table names are specified, then only these tables will be affected.

Return dict result `dict_keys(['executions', 'created', 'skipped', 'tables_created', 'tables_skipped'])`

drop_schemas(**tables*) → dict
Drop all tables listed in *SCHEMAS* if they exist.

Parameters **tables** (*str*) – If one or more table names are specified, then only these tables will be affected.

Return dict result `dict_keys(['executions', 'created', 'skipped', 'tables_dropped', 'tables_skipped'])`

drop_table(*table: str*) → bool
Drop the table `table` if it exists. If the table exists, it will be dropped and `True` will be returned.

If the table doesn't exist (thus can't be dropped), `False` will be returned.

drop_tables(**tables*) → List[Tuple[str, bool]]
Drop one or more tables as positional arguments.

Returns a list of tuples containing `(table_name:str, was_dropped:bool,)` :param str tables: One or more table names to drop as positional args :return list drop_results: List of tuples containing `(table_name:str, was_dropped:bool,)`

fetch(*sql: str*, **params*, *fetch='all'*, ***kwparams*) → Union[dict, tuple, List[dict], List[tuple], None]
Similar to *query()* but only returns the fetch results, not the execution object nor cursor.

Example Usage (default query mode)::

```
>>> s = GenericDBWrapper()
>>> user = s.fetch("SELECT * FROM users WHERE id = ?;", [123], fetch='one
↪')
>>> user
(123, 'john', 'doe',)
```

Example Usage (dict query mode):

```
>>> s.query_mode = 'dict'     # Or s = SqliteWrapper(query_mode='dict')
>>> res = s.fetch("SELECT * FROM users WHERE id = ?;", [123], fetch='one')
>>> res
{'id': 123, 'first_name': 'john', 'last_name': 'doe'}
```

Parameters

- **fetch** (*str*) – Either `'all'` or `'one'` - controls whether the result is fetched with `GenericCursor.fetchall()` or `GenericCursor.fetchone()`
- **sql** (*str*) – An SQL query to execute on the current DB, as a string.
- **params** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`
- **kwparams** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`

Returns

fetchall(*sql: str*, **params*, ***kwparams*) → Union[List[dict], List[tuple], None]
Alias for *fetch()* with `fetch='all'`

fetchone(*sql: str*, **params*, ***kwparams*) → Union[dict, tuple, None]
Alias for *fetch()* with `fetch='one'`

get_cursor(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*) → Union[privex.db.base.CursorManager, privex.db.types.GenericCursor]

Create and return a new database cursor object, by default the cursor will be wrapped with *CursorManager* to ensure context management (`with` statements) works regardless of whether the database API supports context managing cursors (e.g. `sqlite` does not support cursor contexts).

For sub-classes, you should override `_get_cursor()`, which returns an actual native DB cursor.

Parameters

- **cursor_name** (*str*) – (If DB API supports it) The name for this cursor
- **cursor_class** (*type*) – (If DB API supports it) The cursor class to use

Key bool cursor_mgr (Default: `True`) If True, wrap the returned cursor with *CursorManager*

Key callable close_callback (Default: `None`) Passed onto *CursorManager*

Return GenericCursor cursor A cursor object which should implement at least the basic Python DB API Cursor functionality as specified in *GenericCursor* ((PEP 249)

insert(*_table: str*, *_cursor: privex.db.types.GenericCursor = None*, ***fields*) → Union[privex.helpers.collections.DictObject, privex.db.types.GenericCursor]

Builds and executes an insert query into the table `_table` using the keyword arguments for column names and values.

```
>>> db = GenericDBWrapper(db='SomeDB')
>>> cur = db.insert('users', first_name='John', last_name='Doe', phone='+1-
↪800-123-4567')
>>> cur.lastrowid
15
```

Parameters

- **_table** (*str*) – The table to insert into
- **_cursor** (GenericCursor) – Optionally, specify a cursor to use, instead of the default *cursor*
- **fields** – Keyword args mapping column names to values

Return DictObject cur If no custom cursor was specified, the cursor used to execute the query is converted into a `DictObject` before closing it, then the dict is returned.

Return GenericCursor cur If a custom cursor (`_cursor`) was specified, then that cursor will NOT be auto-closed, and the original cursor instance will be returned.

list_tables() → List[str]

Get a list of tables present in the current database.

Example:

```
>>> GenericDBWrapper().list_tables()
['sqlite_sequence', 'nodes', 'node_api', 'node_failures']
```

Return List[str] tables A list of tables in the database

make_connection(**args*, ***kwargs*) → privex.db.types.GenericConnection

Creates a database connection using `connector_func`, passing all arguments/kwargs directly to the connector function.

Return GenericConnection conn A database connection object, which should implement at least the basic connection object methods as specified in the Python DB API (PEP 249), and in the Protocol type class *GenericConnection*

query(*sql: str*, **params*, *fetch='all'*, ***kwparams*) → Tuple[Optional[iter], Any, privex.db.types.GenericCursor]

Create an instance of your database wrapper:

```
>>> db = GenericDBWrapper()
```

Querying with prepared SQL queries and returning a single row:

```
>>> res, res_exec, cur = db.query("SELECT * FROM users WHERE first_name = ?;",
↪ ['John'], fetch='one')
>>> res
(12, 'John', 'Doe', '123 Example Road',)
>>> cur.close()
```

Querying with plain SQL queries, using query_mode, handling an iterator result, and using the cursor

If your database API returns rows as ``tuple``s or ``list``s, you can use ``query_mode='dict'`` (or set *query_mode* in the constructor) to convert any row results into dictionaries which map each column to their values.

```
>>> res, _, cur = db.query("SELECT * FROM users;", fetch='all', query_mode=
↪'dict')
```

When querying with `fetch='all'`, depending on your database API, `res` may be an iterator, and cannot be accessed via an index like `res[0]`.

You should make sure to iterate the rows using a `for` loop:

```
>>> for row in res:
...     print(row['first_name'], ':', row)
John : {'first_name': 'John', 'last_name': 'Doe', 'id': 12}
Dave : {'first_name': 'Dave', 'last_name': 'Johnson', 'id': 13}
Aaron : {'first_name': 'Aaron', 'last_name': 'Swartz', 'id': 14}
```

Or, if the result object is a generator, then you can auto-iterate the results into a list using `x = list(res)`:

```
>>> rows = list(res)
>>> rows[0]
{'first_name': 'John', 'last_name': 'Doe', 'id': 12}
```

Using the returned cursor (third return item), we can access various metadata about our query. Note that cursor objects vary between database APIs, and not all methods/attributes may be available, or may return different data than shown below:

```
>>> cur.description  # cursor.description often contains the column names␣
↪matching the query columns
(('id', None, None, None, None, None, None), ('first_name', None, None, None,␣
↪None, None, None),
 ('last_name', None, None, None, None, None, None))

>>> _, _, cur = db.query("INSERT INTO users (first_name, last_name) VALUES ('a
↪', 'b')", fetch='no')
```

(continues on next page)

(continued from previous page)

```
>>> cur.rowcount   # cursor.rowcount tells us how many rows were affected by␣
↪a query
1
>>> cur.lastrowid  # cursor.lastrowid tells us the ID of the last row we␣
↪inserted with this cursor
3
```

Parameters

- **sql** (*str*) – An SQL query to execute
- **params** – Any positional arguments other than `sql` will be passed to `cursor.execute`.
- **fetch** (*str*) – Fetch mode. Either `all` (return `cursor.fetchall()` as first return arg), `one` (return `cursor.fetchone()`), or `no` (do not fetch. first return arg is None).
- **kwparams** – Any keyword arguments that aren't specified as parameters / keyword args for this method will be forwarded to `cursor.execute`

Key GenericCursor cursor Use this specific cursor instead of automatically obtaining one

Key cursor_name If your database API supports named cursors (e.g. PostgreSQL), then you may specify `cursor_name` as a keyword argument to use a named cursor for this query.

Key query_mode Either `flat` (fetch results as they were originally returned from the DB), or `dict` (use `_zip_cols()` to convert tuple/list rows into dicts mapping col:value).

Return iter results (tuple item 1) An iterable such as a generator, or storage type e.g. `list` or `dict`. **NOTE:** If you've set `fetch='all'`, depending on your database adapter, this may be a generator or other form of iterator that cannot be directly accessed via index (i.e. `res[123]`). Instead you must iterate it with a `for` loop, or cast it into a list/tuple to automatically iterate it into an indexed object, e.g. ``list(res)`

Return Any res_exec (tuple item 2) The object returned from running `cur.execute(sql, *params, **kwparams)`. This may be a cursor, but may also vary based on database API.

Return GenericCursor cur (tuple item 3) The cursor that was used to execute and fetch your query. To allow for use with server side cursors, the cursor is NOT closed automatically. To avoid stale cursors, it's best to run `cur.close()` when you're done with handling the returned results.

query_mode: str = None

Per-instance attribute, either:

- `'flat'` (*query()* returns rows as tuples)
- `'dict'` (*query()* returns rows as dicts mapping column names to values)

recreate_schemas(**tables*) → dict

Drop all tables then re-create them.

Shortcut for running *drop_schemas()* followed by *create_schemas()*.

Parameters **tables** (*str*) – If one or more table names are specified, then only these tables will be affected.

Return dict result `dict_keys(['tables_created', 'skipped_create', 'skipped_drop', 'tables_dropped'])`

table_exists(*table: str*) → bool

Returns `True` if the table `table` exists in the database, otherwise `False`.

```
>>> GenericDBWrapper().table_exists('some_table')
True
>>> GenericDBWrapper().table_exists('other_table')
False
```

Parameters **table** (*str*) – The table to check for existence.

Return bool exists `True` if the table `table` exists in the database, otherwise `False`.

tables_created: Set[str] = {}

This is a static class attribute which tracks which tables have already been created, avoiding *create_schemas()* having to make as many queries every time the class is constructed.

2.3.3.1 Methods

Methods

__init__([db, connector_func])	Initialise the database wrapper class.
action(sql, *params, **kwparams)	Use *action()* as a simple alias method for running "action" queries which don't return results, only affected row counts.
close_cursor()	
create_schema(table[, schema])	Create the individual table `table`, either uses the create statement `schema`, or if `schema` is empty, then checks for a pre-existing CREATE statement for `table` in *SCHEMAS*.
create_schemas(*tables)	Create all tables listed in *SCHEMAS* if they don't already exist.
drop_schemas(*tables)	Drop all tables listed in *SCHEMAS* if they exist.
drop_table(table)	Drop the table `table` if it exists.
drop_tables(*tables)	Drop one or more tables as positional arguments.
fetch(sql, *params[, fetch])	Similar to *query()* but only returns the fetch results, not the execution object nor cursor.
fetchall(sql, *params, **kwparams)	Alias for *fetch()* with `fetch='all'`
fetchone(sql, *params, **kwparams)	Alias for *fetch()* with `fetch='one'`
get_cursor([cursor_name, cursor_class])	Create and return a new database cursor object, by default the cursor will be wrapped with *CursorManager* to ensure context management (`with` statements) works regardless of whether the database API supports context managing cursors (e.g.
list_tables()	Get a list of tables present in the current database.
make_connection(*args, **kwargs)	Creates a database connection using `connector_func`, passing all arguments/kwargs directly to the connector function.
query(sql, *params[, fetch])	Create an instance of your database wrapper:
recreate_schemas(*tables)	Drop all tables then re-create them.
table_exists(table)	Returns `True` if the table `table` exists in the database, otherwise `False`.

2.3.3.1.1 __init__

GenericDBWrapper.**__init__**(*db=None*, *connector_func: callable = None*, ***kwargs*)

Initialise the database wrapper class.

This constructor sets `_conn` to None, and sets up various instance variables such as `connector_func` and `query_mode`.

While you can set various instance variables such as `query_mode` via this constructor, if you're inheriting this class, it's recommended that you override the `DEFAULT_` static class attributes to your preference.

Parameters

- **db** (*str*) – The name / path of the database to connect to
- **connector_func** (*callable*) – A function / method / lambda which returns a database connection object which acts like *GenericConnection*

Key bool auto_create_schema (Default: `True`) If True, call *create_schemas()* during constructor.

Key list connector_args A list of arguments to be passed as positional args to `connector_func`

Key dict connector_kwargs A dict of arguments to passed as keyword args to `connector_func`

Key str query_mode Either `flat` (return tuples) or `dict` (return dicts of column:value) Controls how results are returned from query functions, e.g. *query()* and *fetch()*

Key str table_query The query used to check for existence of a table. The query should take one prepared statement argument (the table name to check for), and the first column returned must be named `table_count` - an integer containing how many tables were found under the given name (usually just 0 if not found, 1 if found).

Key str table_list_query The query used to obtain a list of tables in the database. The query should take no arguments, and return rows containing one column each, `name` - the name of the table.

2.3.3.1.2 action

GenericDBWrapper.**action**(*sql: str*, **params*, ***kwparams*) → int

Use *action()* as a simple alias method for running "action" queries which don't return results, only affected row counts.

For example `INSERT`, `UPDATE`, `CREATE` etc. queries.

This method calls *query()* with `fetch='no'`, saves the row count into a variable, closes the cursor, then returns the affected row count as an integer.

```
>>> db = GenericDBWrapper('SomeDB')
>>> rows_affected = db.action("DELETE FROM users WHERE first_name LIKE 'John%';")
>>> rows_affected
4
```

Parameters

- **sql** (*str*) – An SQL query to execute on the current DB, as a string.
- **params** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`
- **kwparams** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`

Return int row_count Number of rows affected

2.3.3.1.3 close_cursor

GenericDBWrapper.**close_cursor**()

2.3.3.1.4 create_schema

GenericDBWrapper.**create_schema**(*table: str*, *schema: str = None*)

Create the individual table `table`, either uses the create statement `schema`, or if `schema` is empty, then checks for a pre-existing CREATE statement for `table` in *SCHEMAS*.

```
>>> db = GenericDBWrapper('SomeDBName')
>>> db.create_schema('users', 'CREATE TABLE users (id INT PRIMARY KEY, name␣
↪VARCHAR(200));')
```

Parameters

- **table** –
- **schema** –

Returns

2.3.3.1.5 create_schemas

GenericDBWrapper.**create_schemas**(**tables*) → dict

Create all tables listed in *SCHEMAS* if they don't already exist.

Parameters **tables** (*str*) – If one or more table names are specified, then only these tables will be affected.

Return dict result `dict_keys(['executions', 'created', 'skipped', 'tables_created', 'tables_skipped'])`

2.3.3.1.6 drop_schemas

GenericDBWrapper.**drop_schemas**(**tables*) → dict

Drop all tables listed in *SCHEMAS* if they exist.

Parameters **tables** (*str*) – If one or more table names are specified, then only these tables will be affected.

Return dict result `dict_keys(['executions', 'created', 'skipped', 'tables_dropped', 'tables_skipped'])`

2.3.3.1.7 drop_table

`GenericDBWrapper.`**`drop_table`**(*table: str*) → bool

Drop the table `table` if it exists. If the table exists, it will be dropped and `True` will be returned.

If the table doesn't exist (thus can't be dropped), `False` will be returned.

2.3.3.1.8 drop_tables

`GenericDBWrapper.`**`drop_tables`**(**tables*) → List[Tuple[str, bool]]

Drop one or more tables as positional arguments.

Returns a list of tuples containing `(table_name:str, was_dropped:bool,)` :param str tables: One or more table names to drop as positional args :return list drop_results: List of tuples containing `(table_name:str, was_dropped:bool,)`

2.3.3.1.9 fetch

`GenericDBWrapper.`**`fetch`**(*sql: str*, **params*, *fetch='all'*, ***kwparams*) → Union[dict, tuple, List[dict], List[tuple], None]

Similar to *`query()`* but only returns the fetch results, not the execution object nor cursor.

Example Usage (default query mode)::

```
>>> s = GenericDBWrapper()
>>> user = s.fetch("SELECT * FROM users WHERE id = ?;", [123], fetch='one')
>>> user
(123, 'john', 'doe',)
```

Example Usage (dict query mode):

```
>>> s.query_mode = 'dict'     # Or s = SqliteWrapper(query_mode='dict')
>>> res = s.fetch("SELECT * FROM users WHERE id = ?;", [123], fetch='one')
>>> res
{'id': 123, 'first_name': 'john', 'last_name': 'doe'}
```

Parameters

- **`fetch`** (*`str`*) – Either `'all'` or `'one'` - controls whether the result is fetched with `GenericCursor.fetchall()` or `GenericCursor.fetchone()`
- **`sql`** (*`str`*) – An SQL query to execute on the current DB, as a string.
- **`params`** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`
- **`kwparams`** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`

Returns

2.3.3.1.10 fetchall

GenericDBWrapper.**fetchall**(*sql: str*, **params*, ***kwparams*) → Union[List[dict], List[tuple], None]
Alias for *fetch()* with `fetch='all'`

2.3.3.1.11 fetchone

GenericDBWrapper.**fetchone**(*sql: str*, **params*, ***kwparams*) → Union[dict, tuple, None]
Alias for *fetch()* with `fetch='one'`

2.3.3.1.12 get_cursor

GenericDBWrapper.**get_cursor**(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*) → Union[privex.db.base.CursorManager, privex.db.types.GenericCursor]
Create and return a new database cursor object, by default the cursor will be wrapped with *CursorManager* to ensure context management (`with` statements) works regardless of whether the database API supports context managing cursors (e.g. `sqlite` does not support cursor contexts).

For sub-classes, you should override `_get_cursor()`, which returns an actual native DB cursor.

Parameters

- **cursor_name** (*str*) – (If DB API supports it) The name for this cursor
- **cursor_class** (*type*) – (If DB API supports it) The cursor class to use

Key bool cursor_mgr (Default: `True`) If True, wrap the returned cursor with *CursorManager*

Key callable close_callback (Default: `None`) Passed onto *CursorManager*

Return GenericCursor cursor A cursor object which should implement at least the basic Python DB API Cursor functionality as specified in *GenericCursor* ((PEP 249)

2.3.3.1.13 list_tables

GenericDBWrapper.**list_tables**() → List[str]
Get a list of tables present in the current database.

Example:

```
>>> GenericDBWrapper().list_tables()
['sqlite_sequence', 'nodes', 'node_api', 'node_failures']
```

Return List[str] tables A list of tables in the database

2.3.3.1.14 make_connection

GenericDBWrapper.**make_connection**(**args*, ***kwargs*) → privex.db.types.GenericConnection

Creates a database connection using `connector_func`, passing all arguments/kwargs directly to the connector function.

Return GenericConnection conn A database connection object, which should implement at least the basic connection object methods as specified in the Python DB API (PEP 249), and in the Protocol type class *GenericConnection*

2.3.3.1.15 query

GenericDBWrapper.**query**(*sql: str*, **params*, *fetch='all'*, ***kwparams*) → Tuple[Optional[iter], Any, privex.db.types.GenericCursor]

Create an instance of your database wrapper:

```
>>> db = GenericDBWrapper()
```

Querying with prepared SQL queries and returning a single row:

```
>>> res, res_exec, cur = db.query("SELECT * FROM users WHERE first_name = ?;", [
↪'John'], fetch='one')
>>> res
(12, 'John', 'Doe', '123 Example Road',)
>>> cur.close()
```

Querying with plain SQL queries, using query_mode, handling an iterator result, and using the cursor

If your database API returns rows as ``tuple``s or ``list``s, you can use ``query_mode='dict'`` (or set *query_mode* in the constructor) to convert any row results into dictionaries which map each column to their values.

```
>>> res, _, cur = db.query("SELECT * FROM users;", fetch='all', query_mode='dict')
```

When querying with `fetch='all'`, depending on your database API, `res` may be an iterator, and cannot be accessed via an index like `res[0]`.

You should make sure to iterate the rows using a `for` loop:

```
>>> for row in res:
...     print(row['first_name'], ':', row)
John : {'first_name': 'John', 'last_name': 'Doe', 'id': 12}
Dave : {'first_name': 'Dave', 'last_name': 'Johnson', 'id': 13}
Aaron : {'first_name': 'Aaron', 'last_name': 'Swartz', 'id': 14}
```

Or, if the result object is a generator, then you can auto-iterate the results into a list using `x = list(res)`:

```
>>> rows = list(res)
>>> rows[0]
{'first_name': 'John', 'last_name': 'Doe', 'id': 12}
```

Using the returned cursor (third return item), we can access various metadata about our query. Note that cursor objects vary between database APIs, and not all methods/attributes may be available, or may return different data than shown below:

```
>>> cur.description  # cursor.description often contains the column names␣
↪matching the query columns
(('id', None, None, None, None, None, None), ('first_name', None, None, None,␣
↪None, None, None),
 ('last_name', None, None, None, None, None, None))

>>> _, _, cur = db.query("INSERT INTO users (first_name, last_name) VALUES ('a',
↪'b')", fetch='no')
>>> cur.rowcount   # cursor.rowcount tells us how many rows were affected by a␣
↪query
1
>>> cur.lastrowid  # cursor.lastrowid tells us the ID of the last row we inserted␣
↪with this cursor
3
```

Parameters

- **sql** (*str*) – An SQL query to execute
- **params** – Any positional arguments other than `sql` will be passed to `cursor.execute`.
- **fetch** (*str*) – Fetch mode. Either `all` (return `cursor.fetchall()` as first return arg), `one` (return `cursor.fetchone()`), or `no` (do not fetch. first return arg is None).
- **kwparams** – Any keyword arguments that aren't specified as parameters / keyword args for this method will be forwarded to `cursor.execute`

Key GenericCursor cursor Use this specific cursor instead of automatically obtaining one

Key cursor_name If your database API supports named cursors (e.g. PostgreSQL), then you may specify `cursor_name` as a keyword argument to use a named cursor for this query.

Key query_mode Either `flat` (fetch results as they were originally returned from the DB), or `dict` (use `_zip_cols()` to convert tuple/list rows into dicts mapping col:value).

Return iter results (tuple item 1) An iterable such as a generator, or storage type e.g. `list` or `dict`. **NOTE:** If you've set `fetch='all'`, depending on your database adapter, this may be a generator or other form of iterator that cannot be directly accessed via index (i.e. `res[123]`). Instead you must iterate it with a `for` loop, or cast it into a list/tuple to automatically iterate it into an indexed object, e.g. ``list(res)`

Return Any res_exec (tuple item 2) The object returned from running `cur.execute(sql, *params, **kwparams)`. This may be a cursor, but may also vary based on database API.

Return GenericCursor cur (tuple item 3) The cursor that was used to execute and fetch your query. To allow for use with server side cursors, the cursor is NOT closed automatically. To avoid stale cursors, it's best to run `cur.close()` when you're done with handling the returned results.

2.3.3.1.16 recreate_schemas

`GenericDBWrapper.`**`recreate_schemas`**(**tables*) → dict

Drop all tables then re-create them.

Shortcut for running *drop_schemas()* followed by *create_schemas()*.

Parameters **tables** (*str*) – If one or more table names are specified, then only these tables will be affected.

Return dict result `dict_keys(['tables_created', 'skipped_create', 'skipped_drop', 'tables_dropped'])`

2.3.3.1.17 table_exists

GenericDBWrapper.**table_exists**(*table: str*) → bool

Returns `True` if the table `table` exists in the database, otherwise `False`.

```
>>> GenericDBWrapper().table_exists('some_table')
True
>>> GenericDBWrapper().table_exists('other_table')
False
```

Parameters **table** (*str*) – The table to check for existence.

Return bool exists `True` if the table `table` exists in the database, otherwise `False`.

2.3.3.2 Attributes

Attributes

AUTO_ZIP_COLS	If your database API doesn't support returning rows as dicts or a dict-like structure (e.g.
DEFAULT_ENABLE_EXECUTION_LOG	
DEFAULT_QUERY_MODE	This attribute can be overridden on your inheriting wrapper class to change the default query mode used if one isn't specified in the constructor.
DEFAULT_TABLE_LIST_QUERY	
DEFAULT_TABLE_QUERY	
SCHEMAS	This should be set as a class attribute to a list of two value tuples, each containing the name of a table, and the SQL query to create the table if it doesn't exist.
conn	Get or create a database connection
cursor	
tables_created	This is a static class attribute which tracks which tables have already been created, avoiding *create_schemas()* having to make as many queries every time the class is constructed.

2.3.3.2.1 AUTO_ZIP_COLS

GenericDBWrapper.**AUTO_ZIP_COLS: bool = True**

If your database API doesn't support returning rows as dicts or a dict-like structure (e.g. sqlite3), then when this setting is enabled, `_zip_cols()` will be called to zip each row with the result column names into a dictionary.

If your database API supports returning rows as a dictionary either by default, or via a cursor/connection class (e.g. PostgreSQL with psycopg2) then you should set this to `False` and use the cursor/connection class instead.

2.3.3.2.2 DEFAULT_ENABLE_EXECUTION_LOG

GenericDBWrapper.**DEFAULT_ENABLE_EXECUTION_LOG: bool = True**

2.3.3.2.3 DEFAULT_QUERY_MODE

GenericDBWrapper.**DEFAULT_QUERY_MODE: str = 'dict'**

This attribute can be overridden on your inheriting wrapper class to change the default query mode used if one isn't specified in the constructor.

2.3.3.2.4 DEFAULT_TABLE_LIST_QUERY

GenericDBWrapper.**DEFAULT_TABLE_LIST_QUERY = ''**

2.3.3.2.5 DEFAULT_TABLE_QUERY

GenericDBWrapper.**DEFAULT_TABLE_QUERY = ''**

2.3.3.2.6 SCHEMAS

GenericDBWrapper.**SCHEMAS: List[Tuple[str, str]] = []**

This should be set as a class attribute to a list of two value tuples, each containing the name of a table, and the SQL query to create the table if it doesn't exist.

Example:

```
SCHEMAS = [
    (
        'my_table',
        "CREATE TABLE my_table (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT);"
    ),
    (
        'other_table',
        "CREATE TABLE other_table (id INTEGER PRIMARY KEY AUTOINCREMENT, example␣
↪TEXT);"
    ),
]
```

2.3.3.2.7 conn

property GenericDBWrapper.**conn**

Get or create a database connection

2.3.3.2.8 cursor

property GenericDBWrapper.**cursor**

2.3.3.2.9 tables_created

GenericDBWrapper.**tables_created: Set[str] = {}**

This is a static class attribute which tracks which tables have already been created, avoiding *create_schemas()* having to make as many queries every time the class is constructed.

class privex.db.base.**AsyncCursorManager**(*cursor: CUR*, *close_callback: Optional[Type[Coroutine[Any, Any, None]]] = None*)

Async version of *CursorManager*

Not all database API's support context management with their cursors, so this class wraps a given database cursor objects, and provides context management methods `__enter__()` and `__exit__()`

can_cleanup: bool = None

This becomes True if this is the **first** context manager instance for a cursor

is_context_manager: bool = None

`True` if this class is being used in a `with` statement, otherwise `False`.

class privex.db.base.**CursorManager**(*cursor: CUR*, *close_callback: Optional[callable] = None*)

Not all database API's support context management with their cursors, so this class wraps a given database cursor objects, and provides context management methods `__enter__()` and `__exit__()`

can_cleanup: bool = None

This becomes True if this is the **first** context manager instance for a cursor

is_context_manager: bool = None

`True` if this class is being used in a `with` statement, otherwise `False`.

class privex.db.base.**DBExecution**(**args*, ***kwargs*)

class privex.db.base.**GenericAsyncDBWrapper**(*db=None*, *connector_func: callable = None*, ***kwargs*)

conn

Get or create a database connection

drop_table(*table: str*) → Union[bool, Coroutine[Any, Any, bool]]

Drop the table `table` if it exists. If the table exists, it will be dropped and `True` will be returned.

If the table doesn't exist (thus can't be dropped), `False` will be returned.

async get_cursor(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*) → Union[privex.db.base.AsyncCursorManager, privex.db.types.GenericAsyncCursor]

Create and return a new database cursor object, by default the cursor will be wrapped with *CursorManager* to ensure context management (`with` statements) works regardless of whether the database API supports context managing cursors (e.g. `sqlite` does not support cursor contexts).

For sub-classes, you should override `_get_cursor()`, which returns an actual native DB cursor.

Parameters

- **cursor_name** (*str*) – (If DB API supports it) The name for this cursor
- **cursor_class** (*type*) – (If DB API supports it) The cursor class to use

Key bool cursor_mgr (Default: `True`) If True, wrap the returned cursor with *CursorManager*

Key callable close_callback (Default: `None`) Passed onto *CursorManager*

Return GenericCursor cursor A cursor object which should implement at least the basic Python DB API Cursor functionality as specified in *GenericCursor* ((PEP 249)

class privex.db.base.**GenericDBWrapper**(*db=None*, *connector_func: callable = None*, ***kwargs*)

This is a generic database wrapper class, which implements various methods such as:

- Querying methods such as *query()*, *fetch()*, *fetchone()*, *fetchall()*
- Table management functions such as *create_schemas()*, *drop_schemas()* and *drop_table()*
- Connection and cursor methods / properties: *conn*, *get_cursor()*

While this class is intended to be subclassed by DBMS-specific wrapper classes, all methods follow the Python DB API (PEP 249) and the ANSI SQL standard, meaning very little modification is actually required to adapt this wrapper to most database systems.

See *PostgresWrapper* and *SqliteWrapper* for implementation examples.

AUTO_ZIP_COLS: bool = True

If your database API doesn't support returning rows as dicts or a dict-like structure (e.g. sqlite3), then when this setting is enabled, `_zip_cols()` will be called to zip each row with the result column names into a dictionary.

If your database API supports returning rows as a dictionary either by default, or via a cursor/connection class (e.g. PostgreSQL with psycopg2) then you should set this to `False` and use the cursor/connection class instead.

DEFAULT_PLACEHOLDER: str = '?'

The placeholder used by the database API for prepared statements in .execute()

DEFAULT_QUERY_MODE: str = 'dict'

This attribute can be overridden on your inheriting wrapper class to change the default query mode used if one isn't specified in the constructor.

SCHEMAS: List[Tuple[str, str]] = []

This should be set as a class attribute to a list of two value tuples, each containing the name of a table, and the SQL query to create the table if it doesn't exist.

Example:

```
SCHEMAS = [
    (
        'my_table',
        "CREATE TABLE my_table (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"
    ),
    (
        'other_table',
        "CREATE TABLE other_table (id INTEGER PRIMARY KEY AUTOINCREMENT,␣
↪example TEXT);"
    ),
]
```

action(*sql: str*, **params*, ***kwparams*) → int

Use *action()* as a simple alias method for running "action" queries which don't return results, only affected row counts.

For example `INSERT`, `UPDATE`, `CREATE` etc. queries.

This method calls *`query()`* with `fetch='no'`, saves the row count into a variable, closes the cursor, then returns the affected row count as an integer.

```
>>> db = GenericDBWrapper('SomeDB')
>>> rows_affected = db.action("DELETE FROM users WHERE first_name LIKE 'John%
↪';")
>>> rows_affected
4
```

Parameters

- **`sql`** (*`str`*) – An SQL query to execute on the current DB, as a string.
- **`params`** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`
- **`kwparams`** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`

Return int row_count Number of rows affected

`property conn`

Get or create a database connection

`create_schema`(*table: str*, *schema: str = None*)

Create the individual table `table`, either uses the create statement `schema`, or if `schema` is empty, then checks for a pre-existing CREATE statement for `table` in *`SCHEMAS`*.

```
>>> db = GenericDBWrapper('SomeDBName')
>>> db.create_schema('users', 'CREATE TABLE users (id INT PRIMARY KEY, name␣
↪VARCHAR(200));')
```

Parameters

- **`table`** –
- **`schema`** –

Returns

`create_schemas`(**tables*) → dict

Create all tables listed in *`SCHEMAS`* if they don't already exist.

Parameters **`tables`** (*`str`*) – If one or more table names are specified, then only these tables will be affected.

Return dict result `dict_keys(['executions', 'created', 'skipped', 'tables_created', 'tables_skipped'])`

`drop_schemas`(**tables*) → dict

Drop all tables listed in *`SCHEMAS`* if they exist.

Parameters **`tables`** (*`str`*) – If one or more table names are specified, then only these tables will be affected.

Return dict result `dict_keys(['executions', 'created', 'skipped', 'tables_dropped', 'tables_skipped'])`

drop_table(*table: str*) → bool
Drop the table `table` if it exists. If the table exists, it will be dropped and `True` will be returned.

If the table doesn't exist (thus can't be dropped), `False` will be returned.

drop_tables(**tables*) → List[Tuple[str, bool]]
Drop one or more tables as positional arguments.

Returns a list of tuples containing `(table_name:str, was_dropped:bool,)` :param str tables: One or more table names to drop as positional args :return list drop_results: List of tuples containing `(table_name:str, was_dropped:bool,)`

fetch(*sql: str*, **params*, *fetch='all'*, ***kwparams*) → Union[dict, tuple, List[dict], List[tuple], None]
Similar to *query()* but only returns the fetch results, not the execution object nor cursor.

Example Usage (default query mode)::

```
>>> s = GenericDBWrapper()
>>> user = s.fetch("SELECT * FROM users WHERE id = ?;", [123], fetch='one
↪')
>>> user
(123, 'john', 'doe',)
```

Example Usage (dict query mode):

```
>>> s.query_mode = 'dict'     # Or s = SqliteWrapper(query_mode='dict')
>>> res = s.fetch("SELECT * FROM users WHERE id = ?;", [123], fetch='one')
>>> res
{'id': 123, 'first_name': 'john', 'last_name': 'doe'}
```

Parameters

- **fetch** (*str*) – Either `'all'` or `'one'` - controls whether the result is fetched with `GenericCursor.fetchall()` or `GenericCursor.fetchone()`
- **sql** (*str*) – An SQL query to execute on the current DB, as a string.
- **params** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`
- **kwparams** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwparams)`

Returns

fetchall(*sql: str*, **params*, ***kwparams*) → Union[List[dict], List[tuple], None]
Alias for *fetch()* with `fetch='all'`

fetchone(*sql: str*, **params*, ***kwparams*) → Union[dict, tuple, None]
Alias for *fetch()* with `fetch='one'`

get_cursor(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*) → Union[privex.db.base.CursorManager, privex.db.types.GenericCursor]
Create and return a new database cursor object, by default the cursor will be wrapped with *CursorManager* to ensure context management (`with` statements) works regardless of whether the database API supports context managing cursors (e.g. `sqlite` does not support cursor contexts).

For sub-classes, you should override `_get_cursor()`, which returns an actual native DB cursor.

Parameters

- **cursor_name** (*str*) – (If DB API supports it) The name for this cursor

- **cursor_class** (*type*) – (If DB API supports it) The cursor class to use

Key bool cursor_mgr (Default: `True`) If True, wrap the returned cursor with *CursorManager*

Key callable close_callback (Default: `None`) Passed onto *CursorManager*

Return GenericCursor cursor A cursor object which should implement at least the basic Python DB API Cursor functionality as specified in *GenericCursor* ((PEP 249)

insert(*_table: str*, *_cursor: privex.db.types.GenericCursor = None*, ***fields*) → Union[privex.helpers.collections.DictObject, privex.db.types.GenericCursor]

Builds and executes an insert query into the table `_table` using the keyword arguments for column names and values.

```
>>> db = GenericDBWrapper(db='SomeDB')
>>> cur = db.insert('users', first_name='John', last_name='Doe', phone='+1-
↪800-123-4567')
>>> cur.lastrowid
15
```

Parameters

- **_table** (*str*) – The table to insert into
- **_cursor** (GenericCursor) – Optionally, specify a cursor to use, instead of the default *cursor*
- **fields** – Keyword args mapping column names to values

Return DictObject cur If no custom cursor was specified, the cursor used to execute the query is converted into a `DictObject` before closing it, then the dict is returned.

Return GenericCursor cur If a custom cursor (`_cursor`) was specified, then that cursor will NOT be auto-closed, and the original cursor instance will be returned.

list_tables() → List[str]

Get a list of tables present in the current database.

Example:

```
>>> GenericDBWrapper().list_tables()
['sqlite_sequence', 'nodes', 'node_api', 'node_failures']
```

Return List[str] tables A list of tables in the database

make_connection(**args*, ***kwargs*) → privex.db.types.GenericConnection

Creates a database connection using `connector_func`, passing all arguments/kwargs directly to the connector function.

Return GenericConnection conn A database connection object, which should implement at least the basic connection object methods as specified in the Python DB API (PEP 249), and in the Protocol type class *GenericConnection*

query(*sql: str*, **params*, *fetch='all'*, ***kwparams*) → Tuple[Optional[iter], Any, privex.db.types.GenericCursor]

Create an instance of your database wrapper:

```
>>> db = GenericDBWrapper()
```

Querying with prepared SQL queries and returning a single row:

```
>>> res, res_exec, cur = db.query("SELECT * FROM users WHERE first_name = ?;",
↪ ['John'], fetch='one')
>>> res
(12, 'John', 'Doe', '123 Example Road',)
>>> cur.close()
```

Querying with plain SQL queries, using query_mode, handling an iterator result, and using the cursor

If your database API returns rows as ``tuple``s or ``list``s, you can use ``query_mode='dict'`` (or set *query_mode* in the constructor) to convert any row results into dictionaries which map each column to their values.

```
>>> res, _, cur = db.query("SELECT * FROM users;", fetch='all', query_mode=
↪'dict')
```

When querying with `fetch='all'`, depending on your database API, `res` may be an iterator, and cannot be accessed via an index like `res[0]`.

You should make sure to iterate the rows using a `for` loop:

```
>>> for row in res:
...     print(row['first_name'], ':', row)
John : {'first_name': 'John', 'last_name': 'Doe', 'id': 12}
Dave : {'first_name': 'Dave', 'last_name': 'Johnson', 'id': 13}
Aaron : {'first_name': 'Aaron', 'last_name': 'Swartz', 'id': 14}
```

Or, if the result object is a generator, then you can auto-iterate the results into a list using `x = list(res)`:

```
>>> rows = list(res)
>>> rows[0]
{'first_name': 'John', 'last_name': 'Doe', 'id': 12}
```

Using the returned cursor (third return item), we can access various metadata about our query. Note that cursor objects vary between database APIs, and not all methods/attributes may be available, or may return different data than shown below:

```
>>> cur.description  # cursor.description often contains the column names␣
↪matching the query columns
(('id', None, None, None, None, None, None), ('first_name', None, None, None,␣
↪None, None, None),
 ('last_name', None, None, None, None, None, None))

>>> _, _, cur = db.query("INSERT INTO users (first_name, last_name) VALUES ('a
↪', 'b')", fetch='no')
>>> cur.rowcount   # cursor.rowcount tells us how many rows were affected by␣
↪a query
1
>>> cur.lastrowid  # cursor.lastrowid tells us the ID of the last row we␣
↪inserted with this cursor
3
```

Parameters

- **sql** (*str*) – An SQL query to execute

- **params** – Any positional arguments other than `sql` will be passed to `cursor.execute`.
- **fetch** (*str*) – Fetch mode. Either `all` (return `cursor.fetchall()` as first return arg), `one` (return `cursor.fetchone()`), or `no` (do not fetch. first return arg is None).
- **kwparams** – Any keyword arguments that aren't specified as parameters / keyword args for this method will be forwarded to `cursor.execute`

Key GenericCursor cursor Use this specific cursor instead of automatically obtaining one

Key cursor_name If your database API supports named cursors (e.g. PostgreSQL), then you may specify `cursor_name` as a keyword argument to use a named cursor for this query.

Key query_mode Either `flat` (fetch results as they were originally returned from the DB), or `dict` (use `_zip_cols()` to convert tuple/list rows into dicts mapping col:value).

Return iter results (tuple item 1) An iterable such as a generator, or storage type e.g. `list` or `dict`. **NOTE:** If you've set `fetch='all'`, depending on your database adapter, this may be a generator or other form of iterator that cannot be directly accessed via index (i.e. `res[123]`). Instead you must iterate it with a `for` loop, or cast it into a list/tuple to automatically iterate it into an indexed object, e.g. ``list(res)`

Return Any res_exec (tuple item 2) The object returned from running `cur.execute(sql, *params, **kwparams)`. This may be a cursor, but may also vary based on database API.

Return GenericCursor cur (tuple item 3) The cursor that was used to execute and fetch your query. To allow for use with server side cursors, the cursor is NOT closed automatically. To avoid stale cursors, it's best to run `cur.close()` when you're done with handling the returned results.

query_mode: str = None

Per-instance attribute, either:

- `'flat'` (*query()* returns rows as tuples)
- `'dict'` (*query()* returns rows as dicts mapping column names to values)

recreate_schemas(**tables*) → dict

Drop all tables then re-create them.

Shortcut for running *drop_schemas()* followed by *create_schemas()*.

Parameters **tables** (*str*) – If one or more table names are specified, then only these tables will be affected.

Return dict result `dict_keys(['tables_created', 'skipped_create', 'skipped_drop', 'tables_dropped'])`

table_exists(*table: str*) → bool

Returns `True` if the table `table` exists in the database, otherwise `False`.

```
>>> GenericDBWrapper().table_exists('some_table')
True
>>> GenericDBWrapper().table_exists('other_table')
False
```

Parameters **table** (*str*) – The table to check for existence.

Return bool exists `True` if the table `table` exists in the database, otherwise `False`.

tables_created: Set[str] = {}
This is a static class attribute which tracks which tables have already been created, avoiding *create_schemas()* having to make as many queries every time the class is constructed.

privex.db.base.**cursor_to_dict**(*cur: Union[privex.db.types.GenericCursor, Any]*) → privex.helpers.collections.DictObject
Convert a cursor object into a dictionary (DictObject), allowing the original cursor to be safely closed without losing any important data.

Parameters **cur** (GenericCursor) – The cursor to extract.

Return DictObject cur_data The cursors attributes extracted into a dictionary (DictObject)

2.4 privex.db.postgres

This module holds *PostgresWrapper* - a somewhat higher level class for interacting with PostgreSQL.

Copyright:

```
    +==================================================+
    |                 © 2019 Privex Inc.               |
    |               https://www.privex.io              |
    +==================================================+
    |                                                  |
    |        Privex's Python Database Library          |
    |        License: X11 / MIT                        |
    |                                                  |
    |        Originally Developed by Privex Inc.       |
    |        Core Developer(s):                        |
    |                                                  |
    |          (+)  Chris (@someguy123) [Privex]       |
    |                                                  |
    +==================================================+

Copyright (c) 2019     Privex Inc.    ( https://www.privex.io )
```

Classes

PostgresWrapper([db, db_user, db_host, db_pass])	Lightweight wrapper class for interacting with PostgreSQL databases.

2.4.1 PostgresWrapper

class privex.db.postgres.**PostgresWrapper**(*db=None, db_user='root', db_host=None, db_pass=None, **kwargs*)
Lightweight wrapper class for interacting with PostgreSQL databases.

Usage

```
class MyManager(PostgresWrapper):
    SCHEMAS: List[Tuple[str, str]] = [
        ('users', "CREATE TABLE users (id SERIAL PRIMARY KEY, name VARCHAR(50));
↪"),
```

(continues on next page)

(continued from previous page)

```
        ('items', "CREATE TABLE items (id INTEGER PRIMARY KEY, name VARCHAR(50));
↪"),
    ]

    def get_items(self):
        return self.fetchall("SELECT * FROM items;")

    def find_item(self, id: int):
        return self.fetchone("SELECT * FROM items WHERE id = %s;", [id]);
```

__init__(*db=None*, *db_user='root'*, *db_host=None*, *db_pass=None*, ***kwargs*)
Initialise the database wrapper class.

Parameters

- **db** (*str*) – Database name
- **db_user** (*str*) – Account username with permission for `db` (defaults to `root`)
- **db_pass** (*str*) – Account password for `db_user` (defaults to `None`)
- **db_host** (*str*) – Database host (defaults to unix socket)

Key str db_schema (Default: `'public'`) Schema used for querying table existence

Key str query_mode Either `'flat'` (query returns tuples) or `'dict'` (query returns dicts). More details in PyDoc block under *query_mode*

Key psycopg2.extensions.cursor cursor_cls If necessary, you may override the Psycopg2 cursor class used by specifying this kwarg. If this isn't specified, *cursor_cls* will default to either `psycopg2.extras.RealDictCursor` if query_mode is `dict`, or `psycopg2.extras.NamedTupleCursor` if query_mode is `flat`.

property conn
Get or create a Postgres connection

db: str = None
PostgreSQL database name

drop_table(*table: str*) → bool
Drop the table `table` if it exists. If the table exists, it will be dropped and `True` will be returned.

If the table doesn't exist (thus can't be dropped), `False` will be returned.

insert(*_table: str*, *_cursor: psycopg2.extensions.cursor = None*, ***fields*) → Union[privex.helpers.collections.DictObject, psycopg2.extensions.cursor]
Builds and executes an insert query into the table `_table` using the keyword arguments for column names and values.

```
>>> db = GenericDBWrapper(db='SomeDB')
>>> cur = db.insert('users', first_name='John', last_name='Doe', phone='+1-
↪800-123-4567')
>>> cur.lastrowid
15
```

Parameters

- **_table** (*str*) – The table to insert into
- **_cursor** (`GenericCursor`) – Optionally, specify a cursor to use, instead of the default *cursor*

- **fields** – Keyword args mapping column names to values

Return DictObject cur If no custom cursor was specified, the cursor used to execute the query is converted into a `DictObject` before closing it, then the dict is returned.

Return GenericCursor cur If a custom cursor (`_cursor`) was specified, then that cursor will NOT be auto-closed, and the original cursor instance will be returned.

last_insert_id(*table_name: str*, *pk_name='id'*)

Get the last inserted ID for a given table + primary key.

Example:

```
>>> db = PostgresWrapper(db='my_db')
>>> db.action('INSERT INTO users (first_name, last_name) VALUES (?, ?);', [
↪'John', 'Doe'])
>>> last_id = db.last_insert_id('users')
>>> db.fetchone('SELECT first_name, last_name FROM users WHERE id = ?', [last_
↪id])
Record(id=16, first_name='John', last_name='Doe')
```

Parameters

- **table_name** (*str*) – The table you want the last insertion for
- **pk_name** (*str*) – The primary key name, e.g. `id`, `username` etc.

Return Any last_id The last `pk_name` inserted into `table_name`

list_tables(*schema: str = None*) → List[str]

Get a list of tables present in the current database.

Example:

```
>>> GenericDBWrapper().list_tables()
['sqlite_sequence', 'nodes', 'node_api', 'node_failures']
```

Return List[str] tables A list of tables in the database

query(*sql: str*, **params*, *fetch='all'*, ***kwparams*) → Tuple[Optional[iter], Any, psycopg2.extensions.cursor]

Create an instance of your database wrapper:

```
>>> db = GenericDBWrapper()
```

Querying with prepared SQL queries and returning a single row:

```
>>> res, res_exec, cur = db.query("SELECT * FROM users WHERE first_name = ?;",
↪ ['John'], fetch='one')
>>> res
(12, 'John', 'Doe', '123 Example Road',)
>>> cur.close()
```

Querying with plain SQL queries, using query_mode, handling an iterator result, and using the cursor

If your database API returns rows as `tuple``s or ``list``s, you can use ``query_mode='dict'` (or set *query_mode* in the constructor) to convert any row results into dictionaries which map each column to their values.

```
>>> res, _, cur = db.query("SELECT * FROM users;", fetch='all', query_mode=
↪'dict')
```

When querying with `fetch='all'`, depending on your database API, `res` may be an iterator, and cannot be accessed via an index like `res[0]`.

You should make sure to iterate the rows using a `for` loop:

```
>>> for row in res:
...     print(row['first_name'], ':', row)
John : {'first_name': 'John', 'last_name': 'Doe', 'id': 12}
Dave : {'first_name': 'Dave', 'last_name': 'Johnson', 'id': 13}
Aaron : {'first_name': 'Aaron', 'last_name': 'Swartz', 'id': 14}
```

Or, if the result object is a generator, then you can auto-iterate the results into a list using `x = list(res)`:

```
>>> rows = list(res)
>>> rows[0]
{'first_name': 'John', 'last_name': 'Doe', 'id': 12}
```

Using the returned cursor (third return item), we can access various metadata about our query. Note that cursor objects vary between database APIs, and not all methods/attributes may be available, or may return different data than shown below:

```
>>> cur.description  # cursor.description often contains the column names␣
↪matching the query columns
(('id', None, None, None, None, None, None), ('first_name', None, None, None,␣
↪None, None, None),
 ('last_name', None, None, None, None, None, None))

>>> _, _, cur = db.query("INSERT INTO users (first_name, last_name) VALUES ('a
↪', 'b')", fetch='no')
>>> cur.rowcount   # cursor.rowcount tells us how many rows were affected by␣
↪a query
1
>>> cur.lastrowid  # cursor.lastrowid tells us the ID of the last row we␣
↪inserted with this cursor
3
```

Parameters

- **sql** (*str*) – An SQL query to execute
- **params** – Any positional arguments other than `sql` will be passed to `cursor.execute`.
- **fetch** (*str*) – Fetch mode. Either `all` (return `cursor.fetchall()` as first return arg), `one` (return `cursor.fetchone()`), or `no` (do not fetch. first return arg is None).
- **kwparams** – Any keyword arguments that aren't specified as parameters / keyword args for this method will be forwarded to `cursor.execute`

Key GenericCursor cursor Use this specific cursor instead of automatically obtaining one

Key cursor_name If your database API supports named cursors (e.g. PostgreSQL), then you may specify `cursor_name` as a keyword argument to use a named cursor for this query.

Key query_mode Either `flat` (fetch results as they were originally returned from the DB), or `dict` (use `_zip_cols()` to convert tuple/list rows into dicts mapping col:value).

Return iter results (tuple item 1) An iterable such as a generator, or storage type e.g. `list` or `dict`. **NOTE:** If you've set `fetch='all'`, depending on your database adapter, this may be a generator or other form of iterator that cannot be directly accessed via index (i.e. `res[123]`). Instead you must iterate it with a `for` loop, or cast it into a list/tuple to automatically iterate it into an indexed object, e.g. ``list(res)`

Return Any res_exec (tuple item 2) The object returned from running `cur.execute(sql, *params, **kwparams)`. This may be a cursor, but may also vary based on database API.

Return GenericCursor cur (tuple item 3) The cursor that was used to execute and fetch your query. To allow for use with server side cursors, the cursor is NOT closed automatically. To avoid stale cursors, it's best to run `cur.close()` when you're done with handling the returned results.

table_exists(*table: str*, *schema: str = None*) → bool

Returns `True` if the table `table` exists in the database, otherwise `False`.

```
>>> GenericDBWrapper().table_exists('some_table')
True
>>> GenericDBWrapper().table_exists('other_table')
False
```

Parameters **table** (*str*) – The table to check for existence.

Return bool exists `True` if the table `table` exists in the database, otherwise `False`.

2.4.1.1 Methods

Methods

__init__([db, db_user, db_host, db_pass])	Initialise the database wrapper class.
builder(table)	
drop_table(table)	Drop the table `table` if it exists.
last_insert_id(table_name[, pk_name])	Get the last inserted ID for a given table + primary key.
list_tables([schema])	Get a list of tables present in the current database.
query(sql, *params[, fetch])	Create an instance of your database wrapper:
table_exists(table[, schema])	Returns `True` if the table `table` exists in the database, otherwise `False`.

2.4.1.1.1 __init__

PostgresWrapper.**__init__**(*db=None*, *db_user='root'*, *db_host=None*, *db_pass=None*, ***kwargs*)

Initialise the database wrapper class.

Parameters

- **db** (*str*) – Database name
- **db_user** (*str*) – Account username with permission for `db` (defaults to `root`)
- **db_pass** (*str*) – Account password for `db_user` (defaults to `None`)
- **db_host** (*str*) – Database host (defaults to unix socket)

Key str db_schema (Default: `'public'`) Schema used for querying table existence

Key str query_mode Either `'flat'` (query returns tuples) or `'dict'` (query returns dicts). More details in PyDoc block under *query_mode*

Key psycopg2.extensions.cursor cursor_cls If necessary, you may override the Psycopg2 cursor class used by specifying this kwarg. If this isn't specified, *cursor_cls* will default to either `psycopg2.extras.RealDictCursor` if query_mode is `dict`, or `psycopg2.extras.NamedTupleCursor` if query_mode is `flat`.

2.4.1.1.2 builder

`PostgresWrapper.`**`builder`**(*table: str*) → privex.db.query.postgres.PostgresQueryBuilder

2.4.1.1.3 drop_table

`PostgresWrapper.`**`drop_table`**(*table: str*) → bool

Drop the table `table` if it exists. If the table exists, it will be dropped and `True` will be returned.

If the table doesn't exist (thus can't be dropped), `False` will be returned.

2.4.1.1.4 last_insert_id

`PostgresWrapper.`**`last_insert_id`**(*table_name: str*, *pk_name='id'*)

Get the last inserted ID for a given table + primary key.

Example:

```
>>> db = PostgresWrapper(db='my_db')
>>> db.action('INSERT INTO users (first_name, last_name) VALUES (?, ?);', ['John',
↪ 'Doe'])
>>> last_id = db.last_insert_id('users')
>>> db.fetchone('SELECT first_name, last_name FROM users WHERE id = ?', [last_id])
Record(id=16, first_name='John', last_name='Doe')
```

Parameters

- **`table_name`** (*str*) – The table you want the last insertion for
- **`pk_name`** (*str*) – The primary key name, e.g. `id`, `username` etc.

Return Any last_id The last `pk_name` inserted into `table_name`

2.4.1.1.5 list_tables

`PostgresWrapper.`**`list_tables`**(*schema: str = None*) → List[str]

Get a list of tables present in the current database.

Example:

```
>>> GenericDBWrapper().list_tables()
['sqlite_sequence', 'nodes', 'node_api', 'node_failures']
```

Return List[str] tables A list of tables in the database

2.4.1.1.6 query

PostgresWrapper.**query**(*sql: str*, **params*, *fetch='all'*, ***kwparams*) → Tuple[Optional[iter], Any, psycopg2.extensions.cursor]

Create an instance of your database wrapper:

```
>>> db = GenericDBWrapper()
```

Querying with prepared SQL queries and returning a single row:

```
>>> res, res_exec, cur = db.query("SELECT * FROM users WHERE first_name = ?;", [
↪'John'], fetch='one')
>>> res
(12, 'John', 'Doe', '123 Example Road',)
>>> cur.close()
```

Querying with plain SQL queries, using query_mode, handling an iterator result, and using the cursor

If your database API returns rows as ``tuple``s or ``list``s, you can use ``query_mode='dict'`` (or set *query_mode* in the constructor) to convert any row results into dictionaries which map each column to their values.

```
>>> res, _, cur = db.query("SELECT * FROM users;", fetch='all', query_mode='dict')
```

When querying with `fetch='all'`, depending on your database API, `res` may be an iterator, and cannot be accessed via an index like `res[0]`.

You should make sure to iterate the rows using a `for` loop:

```
>>> for row in res:
...     print(row['first_name'], ':', row)
John : {'first_name': 'John', 'last_name': 'Doe', 'id': 12}
Dave : {'first_name': 'Dave', 'last_name': 'Johnson', 'id': 13}
Aaron : {'first_name': 'Aaron', 'last_name': 'Swartz', 'id': 14}
```

Or, if the result object is a generator, then you can auto-iterate the results into a list using `x = list(res)`:

```
>>> rows = list(res)
>>> rows[0]
{'first_name': 'John', 'last_name': 'Doe', 'id': 12}
```

Using the returned cursor (third return item), we can access various metadata about our query. Note that cursor objects vary between database APIs, and not all methods/attributes may be available, or may return different data than shown below:

```
>>> cur.description  # cursor.description often contains the column names␣
↪matching the query columns
(('id', None, None, None, None, None, None), ('first_name', None, None, None,␣
↪None, None, None),
 ('last_name', None, None, None, None, None, None))

>>> _, _, cur = db.query("INSERT INTO users (first_name, last_name) VALUES ('a',
↪'b')", fetch='no')
>>> cur.rowcount   # cursor.rowcount tells us how many rows were affected by a␣
↪query
1
```

(continues on next page)

(continued from previous page)

```
>>> cur.lastrowid  # cursor.lastrowid tells us the ID of the last row we inserted␣
↪with this cursor
3
```

Parameters

- **sql** (*str*) – An SQL query to execute
- **params** – Any positional arguments other than `sql` will be passed to `cursor.execute`.
- **fetch** (*str*) – Fetch mode. Either `all` (return `cursor.fetchall()` as first return arg), `one` (return `cursor.fetchone()`), or `no` (do not fetch. first return arg is None).
- **kwparams** – Any keyword arguments that aren't specified as parameters / keyword args for this method will be forwarded to `cursor.execute`

Key GenericCursor cursor Use this specific cursor instead of automatically obtaining one

Key cursor_name If your database API supports named cursors (e.g. PostgreSQL), then you may specify `cursor_name` as a keyword argument to use a named cursor for this query.

Key query_mode Either `flat` (fetch results as they were originally returned from the DB), or `dict` (use `_zip_cols()` to convert tuple/list rows into dicts mapping col:value).

Return iter results (tuple item 1) An iterable such as a generator, or storage type e.g. `list` or `dict`. **NOTE:** If you've set `fetch='all'`, depending on your database adapter, this may be a generator or other form of iterator that cannot be directly accessed via index (i.e. `res[123]`). Instead you must iterate it with a `for` loop, or cast it into a list/tuple to automatically iterate it into an indexed object, e.g. ``list(res)`

Return Any res_exec (tuple item 2) The object returned from running `cur.execute(sql, *params, **kwparams)`. This may be a cursor, but may also vary based on database API.

Return GenericCursor cur (tuple item 3) The cursor that was used to execute and fetch your query. To allow for use with server side cursors, the cursor is NOT closed automatically. To avoid stale cursors, it's best to run `cur.close()` when you're done with handling the returned results.

2.4.1.1.7 table_exists

`PostgresWrapper.`**`table_exists`**(*table: str*, *schema: str = None*) → bool

Returns `True` if the table `table` exists in the database, otherwise `False`.

```
>>> GenericDBWrapper().table_exists('some_table')
True
>>> GenericDBWrapper().table_exists('other_table')
False
```

Parameters **table** (*str*) – The table to check for existence.

Return bool exists `True` if the table `table` exists in the database, otherwise `False`.

2.4.1.2 Attributes

Attributes

AUTO_ZIP_COLS	
DEFAULT_DB	
DEFAULT_QUERY_MODE	
DEFAULT_TABLE_LIST_QUERY	
DEFAULT_TABLE_QUERY	
conn	Get or create a Postgres connection
cursor_cls	
cursor_map	

2.4.1.2.1 AUTO_ZIP_COLS

PostgresWrapper.**AUTO_ZIP_COLS = False**

2.4.1.2.2 DEFAULT_DB

PostgresWrapper.**DEFAULT_DB: str = None**

2.4.1.2.3 DEFAULT_QUERY_MODE

PostgresWrapper.**DEFAULT_QUERY_MODE = 'dict'**

2.4.1.2.4 DEFAULT_TABLE_LIST_QUERY

PostgresWrapper.**DEFAULT_TABLE_LIST_QUERY = "\n SELECT relname as name\n FROM pg_catalog.pg**

2.4.1.2.5 DEFAULT_TABLE_QUERY

PostgresWrapper.**DEFAULT_TABLE_QUERY = "\n SELECT EXISTS (\n SELECT 1\n FROM pg_catalog.pg_**

2.4.1.2.6 conn

property PostgresWrapper.**conn**
Get or create a Postgres connection

2.4.1.2.7 cursor_cls

property PostgresWrapper.**cursor_cls**

2.4.1.2.8 cursor_map

PostgresWrapper.**cursor_map: Dict[str, psycopg2.extensions.cursor] = {'dict': <class 'psyc**

class privex.db.postgres.**PostgresWrapper**(*db=None*, *db_user='root'*, *db_host=None*, *db_pass=None*, ***kwargs*)

Lightweight wrapper class for interacting with PostgreSQL databases.

Usage

```
class MyManager(PostgresWrapper):
    SCHEMAS: List[Tuple[str, str]] = [
        ('users', "CREATE TABLE users (id SERIAL PRIMARY KEY, name VARCHAR(50));
↪"),
        ('items', "CREATE TABLE items (id INTEGER PRIMARY KEY, name VARCHAR(50));
↪"),
    ]

    def get_items(self):
        return self.fetchall("SELECT * FROM items;")

    def find_item(self, id: int):
        return self.fetchone("SELECT * FROM items WHERE id = %s;", [id]);
```

property conn

Get or create a Postgres connection

db: str = None

PostgreSQL database name

drop_table(*table: str*) → bool

Drop the table `table` if it exists. If the table exists, it will be dropped and `True` will be returned.

If the table doesn't exist (thus can't be dropped), `False` will be returned.

insert(*_table: str*, *_cursor: psycopg2.extensions.cursor = None*, ***fields*) → Union[privex.helpers.collections.DictObject, psycopg2.extensions.cursor]

Builds and executes an insert query into the table `_table` using the keyword arguments for column names and values.

```
>>> db = GenericDBWrapper(db='SomeDB')
>>> cur = db.insert('users', first_name='John', last_name='Doe', phone='+1-
↪800-123-4567')
>>> cur.lastrowid
15
```

Parameters

- **_table** (*str*) – The table to insert into
- **_cursor** (GenericCursor) – Optionally, specify a cursor to use, instead of the default *cursor*
- **fields** – Keyword args mapping column names to values

Return DictObject cur If no custom cursor was specified, the cursor used to execute the query is converted into a `DictObject` before closing it, then the dict is returned.

Return GenericCursor cur If a custom cursor (`_cursor`) was specified, then that cursor will NOT be auto-closed, and the original cursor instance will be returned.

last_insert_id(*table_name: str*, *pk_name='id'*)

Get the last inserted ID for a given table + primary key.

Example:

```
>>> db = PostgresWrapper(db='my_db')
>>> db.action('INSERT INTO users (first_name, last_name) VALUES (?, ?);', [
↪'John', 'Doe'])
>>> last_id = db.last_insert_id('users')
>>> db.fetchone('SELECT first_name, last_name FROM users WHERE id = ?', [last_
↪id])
Record(id=16, first_name='John', last_name='Doe')
```

Parameters

- **table_name** (*str*) – The table you want the last insertion for
- **pk_name** (*str*) – The primary key name, e.g. `id`, `username` etc.

Return Any last_id The last `pk_name` inserted into `table_name`

list_tables(*schema: str = None*) → List[str]

Get a list of tables present in the current database.

Example:

```
>>> GenericDBWrapper().list_tables()
['sqlite_sequence', 'nodes', 'node_api', 'node_failures']
```

Return List[str] tables A list of tables in the database

query(*sql: str*, **params*, *fetch='all'*, ***kwparams*) → Tuple[Optional[iter], Any, psycopg2.extensions.cursor]

Create an instance of your database wrapper:

```
>>> db = GenericDBWrapper()
```

Querying with prepared SQL queries and returning a single row:

```
>>> res, res_exec, cur = db.query("SELECT * FROM users WHERE first_name = ?;",
↪ ['John'], fetch='one')
>>> res
(12, 'John', 'Doe', '123 Example Road',)
>>> cur.close()
```

Querying with plain SQL queries, using query_mode, handling an iterator result, and using the cursor

If your database API returns rows as ``tuple``s or ``list``s, you can use ``query_mode='dict'`` (or set *query_mode* in the constructor) to convert any row results into dictionaries which map each column to their values.

```
>>> res, _, cur = db.query("SELECT * FROM users;", fetch='all', query_mode=
↪'dict')
```

When querying with `fetch='all'`, depending on your database API, `res` may be an iterator, and cannot be accessed via an index like `res[0]`.

You should make sure to iterate the rows using a `for` loop:

```
>>> for row in res:
...     print(row['first_name'], ':', row)
John : {'first_name': 'John', 'last_name': 'Doe', 'id': 12}
Dave : {'first_name': 'Dave', 'last_name': 'Johnson', 'id': 13}
Aaron : {'first_name': 'Aaron', 'last_name': 'Swartz', 'id': 14}
```

Or, if the result object is a generator, then you can auto-iterate the results into a list using `x = list(res)`:

```
>>> rows = list(res)
>>> rows[0]
{'first_name': 'John', 'last_name': 'Doe', 'id': 12}
```

Using the returned cursor (third return item), we can access various metadata about our query. Note that cursor objects vary between database APIs, and not all methods/attributes may be available, or may return different data than shown below:

```
>>> cur.description  # cursor.description often contains the column names␣
↪matching the query columns
(('id', None, None, None, None, None, None), ('first_name', None, None, None,␣
↪None, None, None),
 ('last_name', None, None, None, None, None, None))

>>> _, _, cur = db.query("INSERT INTO users (first_name, last_name) VALUES ('a
↪', 'b')", fetch='no')
>>> cur.rowcount   # cursor.rowcount tells us how many rows were affected by␣
↪a query
1
>>> cur.lastrowid  # cursor.lastrowid tells us the ID of the last row we␣
↪inserted with this cursor
3
```

Parameters

- **sql** (*str*) – An SQL query to execute
- **params** – Any positional arguments other than `sql` will be passed to `cursor.execute`.
- **fetch** (*str*) – Fetch mode. Either `all` (return `cursor.fetchall()` as first return arg), `one` (return `cursor.fetchone()`), or `no` (do not fetch. first return arg is None).
- **kwparams** – Any keyword arguments that aren't specified as parameters / keyword args for this method will be forwarded to `cursor.execute`

Key GenericCursor cursor Use this specific cursor instead of automatically obtaining one

Key cursor_name If your database API supports named cursors (e.g. PostgreSQL), then you may specify `cursor_name` as a keyword argument to use a named cursor for this query.

Key query_mode Either `flat` (fetch results as they were originally returned from the DB), or `dict` (use `_zip_cols()` to convert tuple/list rows into dicts mapping col:value).

Return iter results (tuple item 1) An iterable such as a generator, or storage type e.g. `list` or `dict`. **NOTE:** If you've set `fetch='all'`, depending on your database adapter, this may be a generator or other form of iterator that cannot be directly accessed via index (i.e. `res[123]`). Instead you must iterate it with a `for` loop, or cast it into a list/tuple to automatically iterate it into an indexed object, e.g. ``list(res)`

Return Any res_exec (tuple item 2) The object returned from running `cur.execute(sql, *params, **kwparams)`. This may be a cursor, but may also vary based on database API.

Return GenericCursor cur (tuple item 3) The cursor that was used to execute and fetch your query. To allow for use with server side cursors, the cursor is NOT closed automatically. To avoid stale cursors, it's best to run `cur.close()` when you're done with handling the returned results.

table_exists(*table: str*, *schema: str = None*) → bool

Returns `True` if the table `table` exists in the database, otherwise `False`.

```
>>> GenericDBWrapper().table_exists('some_table')
True
>>> GenericDBWrapper().table_exists('other_table')
False
```

Parameters **table** (*str*) – The table to check for existence.

Return bool exists `True` if the table `table` exists in the database, otherwise `False`.

2.5 privex.db.sqlite

This module holds *SqliteWrapper* - a somewhat higher level class for interacting with SQLite3 databases.

Copyright:

```
    +==================================================+
    |                 © 2019 Privex Inc.               |
    |               https://www.privex.io              |
    +==================================================+
    |                                                  |
    |        Privex's Python Database Library          |
    |        License: X11 / MIT                        |
    |                                                  |
    |        Originally Developed by Privex Inc.       |
    |        Core Developer(s):                        |
    |                                                  |
    |          (+)  Chris (@someguy123) [Privex]       |
    |                                                  |
    +==================================================+

Copyright (c) 2019     Privex Inc.    ( https://www.privex.io )
```

Classes

SqliteWrapper([db, isolation_level])	Lightweight wrapper class for interacting with Sqlite3 databases.

2.5.1 SqliteWrapper

class privex.db.sqlite.**SqliteWrapper**(*db: str = None*, *isolation_level=None*, ***kwargs*)

Lightweight wrapper class for interacting with Sqlite3 databases.

Simple direct class usage

```
>>> db_path = expanduser('~/.my_app/my_db.db')
>>> db = SqliteWrapper(db=db_path)
>>> users = db.fetchall("SELECT * FROM users;")
```

Usage

Below is an example wrapper class which uses *SqliteWrapper* as it's parent class.

It overrides the class attributes *DEFAULT_DB_FOLDER*, *DEFAULT_DB_NAME*, and *DEFAULT_DB* - so that if no database path is passed to `MyManager`, then the database file path contained in `MyManager.DEFAULT_DB` will be used as a default.

It also overrides *SCHEMAS* to define 2 tables (`users` and `items`) which will be automatically created when the class is instantiated, unless they already exist.

It adds two methods `get_items` (returns an iterator

```
class MyManager(SqliteWrapper):
    ###
    # If a database path isn't specified, then the class attribute DEFAULT_DB␣
↪will be used.
    ###
    DEFAULT_DB_FOLDER: str = expanduser('~/.my_app')
    DEFAULT_DB_NAME: str = 'my_app.db'
    DEFAULT_DB: str = join(DEFAULT_DB_FOLDER, DEFAULT_DB_NAME)

    ###
    # The SCHEMAS class attribute contains a list of tuples, with each tuple␣
↪containing the name of a
    # table, as well as the SQL query required to create the table if it doesn't␣
↪exist.
    ###
    SCHEMAS: List[Tuple[str, str]] = [
        ('users', "CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"),
        ('items', "CREATE TABLE items (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"),
    ]

    def get_items(self):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchall with a pre-defined SQL query
        return self.fetchall("SELECT * FROM items;")
```

(continues on next page)

(continued from previous page)

```
    def find_item(self, id: int):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchone with a pre-defined SQL query, and interpolates the 'id'␣
↪parameter into
        # the prepared statement.
        return self.fetchone("SELECT * FROM items WHERE id = ?;", [id]);
```

__init__(*db: str = None*, *isolation_level=None*, ***kwargs*)

Parameters

- **db** (*str*) – Relative / absolute path to SQLite3 database file to use.
- **isolation_level** – Isolation level for SQLite3 connection. Defaults to `None` (auto-commit). See the Python SQLite3 Docs for more information.

Key int db_timeout Amount of time to wait for any SQLite3 locks to expire before giving up

Key str query_mode Either `'flat'` (query returns tuples) or `'dict'` (query returns dicts). More details in PyDoc block under *query_mode*

Key bool memory_persist Use a shared in-memory database, which can be accessed by other instances of this class (in this process) - which is cleared after all memory connections are closed. Shortcut for `db='file::memory:?cache=shared'`

DEFAULT_DB: str = '/home/docs/.privex_sqlite/privex_sqlite.db'
Combined *DEFAULT_DB_FOLDER* and *DEFAULT_DB_NAME* used as default absolute path for the sqlite3 database

DEFAULT_DB_FOLDER: str = '/home/docs/.privex_sqlite'
If an absolute path isn't given, store the sqlite3 database file in this folder

DEFAULT_DB_NAME: str = 'privex_sqlite.db'
If no database is specified to *__init__()*, then use this (appended to *DEFAULT_DB_FOLDER*)

property conn
Get or create an SQLite3 connection using DB file *db* and return it

db: str = None
Path to the SQLite3 database for this class instance

insert(*_table: str*, *_cursor: sqlite3.Cursor = None*, ***fields*) → Union[privex.helpers.collections.DictObject, sqlite3.Cursor]
Builds and executes an insert query into the table `_table` using the keyword arguments for column names and values.

```
>>> db = GenericDBWrapper(db='SomeDB')
>>> cur = db.insert('users', first_name='John', last_name='Doe', phone='+1-
↪800-123-4567')
>>> cur.lastrowid
15
```

Parameters

- **_table** (*str*) – The table to insert into
- **_cursor** (*GenericCursor*) – Optionally, specify a cursor to use, instead of the default *cursor*
- **fields** – Keyword args mapping column names to values

Return DictObject cur If no custom cursor was specified, the cursor used to execute the query is converted into a `DictObject` before closing it, then the dict is returned.

Return GenericCursor cur If a custom cursor (`_cursor`) was specified, then that cursor will NOT be auto-closed, and the original cursor instance will be returned.

2.5.1.1 Methods

Methods

__init__([db, isolation_level])	**param str db** Relative / absolute path to SQLite3 database file to use.
builder(table)	

2.5.1.1.1 __init__

`SqliteWrapper.`**`__init__`**(*db: str = None*, *isolation_level=None*, ***kwargs*)

Parameters

- **db** (*str*) – Relative / absolute path to SQLite3 database file to use.
- **isolation_level** – Isolation level for SQLite3 connection. Defaults to `None` (auto-commit). See the Python SQLite3 Docs for more information.

Key int db_timeout Amount of time to wait for any SQLite3 locks to expire before giving up

Key str query_mode Either `'flat'` (query returns tuples) or `'dict'` (query returns dicts). More details in PyDoc block under *query_mode*

Key bool memory_persist Use a shared in-memory database, which can be accessed by other instances of this class (in this process) - which is cleared after all memory connections are closed. Shortcut for `db='file::memory:?cache=shared'`

2.5.1.1.2 builder

`SqliteWrapper.`**`builder`**(*table: str*) → privex.db.query.sqlite.SqliteQueryBuilder

2.5.1.2 Attributes

Attributes

DEFAULT_DB	Combined *DEFAULT_DB_FOLDER* and *DEFAULT_DB_NAME* used as default absolute path for the sqlite3 database
DEFAULT_DB_FOLDER	If an absolute path isn't given, store the sqlite3 database file in this folder
DEFAULT_DB_NAME	If no database is specified to *__init__()*, then use this (appended to *DEFAULT_DB_FOLDER*)

Continued on next page

Table 14 – continued from previous page

DEFAULT_TABLE_LIST_QUERY	
DEFAULT_TABLE_QUERY	
conn	Get or create an SQLite3 connection using DB file *db* and return it

2.5.1.2.1 DEFAULT_DB

SqliteWrapper.**DEFAULT_DB: str = '/home/docs/.privex_sqlite/privex_sqlite.db'**
Combined *DEFAULT_DB_FOLDER* and *DEFAULT_DB_NAME* used as default absolute path for the sqlite3 database

2.5.1.2.2 DEFAULT_DB_FOLDER

SqliteWrapper.**DEFAULT_DB_FOLDER: str = '/home/docs/.privex_sqlite'**
If an absolute path isn't given, store the sqlite3 database file in this folder

2.5.1.2.3 DEFAULT_DB_NAME

SqliteWrapper.**DEFAULT_DB_NAME: str = 'privex_sqlite.db'**
If no database is specified to *__init__()*, then use this (appended to *DEFAULT_DB_FOLDER*)

2.5.1.2.4 DEFAULT_TABLE_LIST_QUERY

SqliteWrapper.**DEFAULT_TABLE_LIST_QUERY = "SELECT name FROM sqlite_master WHERE type = 'tabl**

2.5.1.2.5 DEFAULT_TABLE_QUERY

SqliteWrapper.**DEFAULT_TABLE_QUERY = "SELECT count(name) as table_count FROM sqlite_master W**

2.5.1.2.6 conn

property SqliteWrapper.**conn**
Get or create an SQLite3 connection using DB file *db* and return it

class privex.db.sqlite.**SqliteAsyncWrapper**(*db: str = None, isolation_level=None, **kwargs*)

Usage

Creating an instance:

```
>>> from privex.db import SqliteAsyncWrapper
>>> db = SqliteAsyncWrapper('my_app.db')
```

Inserting rows:

```
>>> db.insert('users', first_name='John', last_name='Doe')
>>> db.insert('users', first_name='Dave', last_name='Johnson')
```

Running raw queries:

```
>>> # fetchone() allows you to run a raw query, and a dict is returned with the␣
↪first row result
>>> row = await db.fetchone("SELECT * FROM users WHERE first_name = ?;", ['John'])
>>> row['first_name']
John
>>> row['last_name']
Doe

>>> # fetchall() runs a query and returns an iterator of the returned rows
>>> rows = await db.fetchall("SELECT * FROM users;")
>>> for user in rows:
...     print(f"First Name: {row['first_name']}   ||    Last Name: {row['last_name
↪']}")
...
First Name: John   ||    Last Name: Doe
First Name: Dave   ||    Last Name: Johnson

>>> # action() is for running queries where you don't want to fetch any results.␣
↪It simply returns the
>>> # affected row count as an integer.
>>> row_count = await db.action('UPDATE users SET first_name = ? WHERE id = ?;', [
↪'David', 2])
>>> print(row_count)
1
```

Creating tables if they don't already exist:

```
>>> # If the table 'users' doesn't exist, the CREATE TABLE query will be executed.
>>> await db.create_schema(
...     'users',
...     "CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT, first_name TEXT,
↪ last_name TEXT);"
... )
>>>
```

Using the query builder:

```
>>> # You can either use it directly
>>> q = db.builder('users')
>>> q.select('first_name', 'last_name').where('first_name', 'John').where_or(
↪'last_name', 'Doe')
>>> results = q.all()
>>> async for row in results:
...     print(f"First Name: {row['first_name']}   ||    Last Name: {row['last_name
↪']}")
...
First Name: John   ||    Last Name: Doe

>>> # Or, you can use it in a ``with`` statement to maintain a singular␣
↪connection, which means you
>>> # can use .fetch_next to fetch a singular row at a time (you can still use .
↪all() and .fetch())
>>> async with db.builder('users') as q:
...     q.select('first_name', 'last_name')
...     row = q.fetch_next()
...     print('Name:', row['first_name'], row['last_name'])    # John Doe
...     row = q.fetch_next()
```

(continues on next page)

(continued from previous page)

```
...     print('Name:', row['first_name'], row['last_name'])    # Dave Johnson
...
Name: John Doe
Name: Dave Johnson
```

Creating a wrapper sub-class of SqliteAsyncWrapper:

```
class MyManager(SqliteAsyncWrapper):
    ###
    # If a database path isn't specified, then the class attribute DEFAULT_DB␣
↪will be used.
    ###
    DEFAULT_DB_FOLDER: str = expanduser('~/.my_app')
    DEFAULT_DB_NAME: str = 'my_app.db'
    DEFAULT_DB: str = join(DEFAULT_DB_FOLDER, DEFAULT_DB_NAME)

    ###
    # The SCHEMAS class attribute contains a list of tuples, with each tuple␣
↪containing the name of a
    # table, as well as the SQL query required to create the table if it doesn't␣
↪exist.
    ###
    SCHEMAS: List[Tuple[str, str]] = [
        ('users', "CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"),
        ('items', "CREATE TABLE items (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"),
    ]

    async def get_items(self):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchall with a pre-defined SQL query
        return await self.fetchall("SELECT * FROM items;")

    async def find_item(self, id: int):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchone with a pre-defined SQL query, and interpolates the 'id'␣
↪parameter into
        # the prepared statement.
        return await self.fetchone("SELECT * FROM items WHERE id = ?;", [id]);
```

DEFAULT_DB: str = '/home/docs/.privex_sqlite/privex_sqlite.db'
Combined *DEFAULT_DB_FOLDER* and *DEFAULT_DB_NAME* used as default absolute path for the sqlite3 database

DEFAULT_DB_FOLDER: str = '/home/docs/.privex_sqlite'
If an absolute path isn't given, store the sqlite3 database file in this folder

DEFAULT_DB_NAME: str = 'privex_sqlite.db'
If no database is specified to *__init__()*, then use this (appended to *DEFAULT_DB_FOLDER*)

conn
Get or create an SQLite3 connection using DB file *db* and return it

db: str = None
Path to the SQLite3 database for this class instance

async get_cursor(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*) → aiosqlite.core.Cursor

Create and return a new database cursor object, by default the cursor will be wrapped with *CursorManager* to ensure context management (`with` statements) works regardless of whether the database API supports context managing cursors (e.g. `sqlite` does not support cursor contexts).

For sub-classes, you should override `_get_cursor()`, which returns an actual native DB cursor.

Parameters

- **cursor_name** (*str*) – (If DB API supports it) The name for this cursor
- **cursor_class** (*type*) – (If DB API supports it) The cursor class to use

Key bool cursor_mgr (Default: `True`) If True, wrap the returned cursor with *CursorManager*

Key callable close_callback (Default: `None`) Passed onto *CursorManager*

Return GenericCursor cursor A cursor object which should implement at least the basic Python DB API Cursor functionality as specified in *GenericCursor* ((PEP 249)

async insert(*_table: str*, *_cursor: aiosqlite.core.Cursor = None*, ***fields*) → Union[privex.helpers.collections.DictObject, aiosqlite.core.Cursor]

Builds and executes an insert query into the table `_table` using the keyword arguments for column names and values.

```
>>> db = GenericDBWrapper(db='SomeDB')
>>> cur = db.insert('users', first_name='John', last_name='Doe', phone='+1-
↪800-123-4567')
>>> cur.lastrowid
15
```

Parameters

- **_table** (*str*) – The table to insert into
- **_cursor** (`GenericCursor`) – Optionally, specify a cursor to use, instead of the default *cursor*
- **fields** – Keyword args mapping column names to values

Return DictObject cur If no custom cursor was specified, the cursor used to execute the query is converted into a `DictObject` before closing it, then the dict is returned.

Return GenericCursor cur If a custom cursor (`_cursor`) was specified, then that cursor will NOT be auto-closed, and the original cursor instance will be returned.

class privex.db.sqlite.**SqliteWrapper**(*db: str = None*, *isolation_level=None*, ***kwargs*)

Lightweight wrapper class for interacting with Sqlite3 databases.

Simple direct class usage

```
>>> db_path = expanduser('~/.my_app/my_db.db')
>>> db = SqliteWrapper(db=db_path)
>>> users = db.fetchall("SELECT * FROM users;")
```

Usage

Below is an example wrapper class which uses *SqliteWrapper* as it's parent class.

It overrides the class attributes *DEFAULT_DB_FOLDER*, *DEFAULT_DB_NAME*, and *DEFAULT_DB* - so that if no database path is passed to `MyManager`, then the database file path contained in `MyManager.DEFAULT_DB` will be used as a default.

It also overrides *SCHEMAS* to define 2 tables (`users` and `items`) which will be automatically created when the class is instantiated, unless they already exist.

It adds two methods `get_items` (returns an iterator

```
class MyManager(SqliteWrapper):
    ###
    # If a database path isn't specified, then the class attribute DEFAULT_DB␣
↪will be used.
    ###
    DEFAULT_DB_FOLDER: str = expanduser('~/.my_app')
    DEFAULT_DB_NAME: str = 'my_app.db'
    DEFAULT_DB: str = join(DEFAULT_DB_FOLDER, DEFAULT_DB_NAME)

    ###
    # The SCHEMAS class attribute contains a list of tuples, with each tuple␣
↪containing the name of a
    # table, as well as the SQL query required to create the table if it doesn't␣
↪exist.
    ###
    SCHEMAS: List[Tuple[str, str]] = [
        ('users', "CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"),
        ('items', "CREATE TABLE items (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"),
    ]

    def get_items(self):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchall with a pre-defined SQL query
        return self.fetchall("SELECT * FROM items;")

    def find_item(self, id: int):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchone with a pre-defined SQL query, and interpolates the 'id'␣
↪parameter into
        # the prepared statement.
        return self.fetchone("SELECT * FROM items WHERE id = ?;", [id]);
```

DEFAULT_DB: str = '/home/docs/.privex_sqlite/privex_sqlite.db'
Combined *DEFAULT_DB_FOLDER* and *DEFAULT_DB_NAME* used as default absolute path for the sqlite3 database

DEFAULT_DB_FOLDER: str = '/home/docs/.privex_sqlite'
If an absolute path isn't given, store the sqlite3 database file in this folder

DEFAULT_DB_NAME: str = 'privex_sqlite.db'
If no database is specified to *__init__()*, then use this (appended to *DEFAULT_DB_FOLDER*)

property conn
Get or create an SQLite3 connection using DB file *db* and return it

db: str = None
Path to the SQLite3 database for this class instance

insert(*_table: str*, *_cursor: sqlite3.Cursor = None*, ***fields*) → Union[privex.helpers.collections.DictObject, sqlite3.Cursor]

Builds and executes an insert query into the table `_table` using the keyword arguments for column names and values.

```
>>> db = GenericDBWrapper(db='SomeDB')
>>> cur = db.insert('users', first_name='John', last_name='Doe', phone='+1-
↪800-123-4567')
>>> cur.lastrowid
15
```

Parameters

- **_table** (*str*) – The table to insert into
- **_cursor** (GenericCursor) – Optionally, specify a cursor to use, instead of the default *cursor*
- **fields** – Keyword args mapping column names to values

Return DictObject cur If no custom cursor was specified, the cursor used to execute the query is converted into a `DictObject` before closing it, then the dict is returned.

Return GenericCursor cur If a custom cursor (`_cursor`) was specified, then that cursor will NOT be auto-closed, and the original cursor instance will be returned.

2.6 privex.db.types

This module holds newly defined types which are used across the module, such as *GenericCursor* and *GenericConnection*

Copyright:

```
    +==================================================+
    |                 © 2019 Privex Inc.               |
    |               https://www.privex.io              |
    +==================================================+
    |                                                  |
    |        Privex's Python Database Library          |
    |        License: X11 / MIT                        |
    |                                                  |
    |        Originally Developed by Privex Inc.       |
    |        Core Developer(s):                        |
    |                                                  |
    |          (+)  Chris (@someguy123) [Privex]       |
    |                                                  |
    +==================================================+

Copyright (c) 2019     Privex Inc.    ( https://www.privex.io )
```

Classes

GenericConnection(*args, **kwargs)	This is a `typing_extensions.Protocol` which represents any database Connection object which follows the Python DB API (PEP 249).
GenericCursor(*args, **kwargs)	This is a `typing_extensions.Protocol` which represents any database Cursor object which follows the Python DB API (PEP 249).

2.6.1 GenericConnection

class privex.db.types.**GenericConnection**(**args*, ***kwargs*)

This is a `typing_extensions.Protocol` which represents any database Connection object which follows the Python DB API (PEP 249).

__init__(**args*, ***kwargs*)

2.6.1.1 Methods

Methods

__init__(*args, **kwargs)
close(*args, **kwargs)
commit(*args, **kwargs)
cursor(*args, **kwargs)
rollback(*args, **kwargs)

2.6.1.1.1 __init__

GenericConnection.**__init__**(**args*, ***kwargs*)

2.6.1.1.2 close

GenericConnection.**close**(**args*, ***kwargs*)

2.6.1.1.3 commit

GenericConnection.**commit**(**args*, ***kwargs*)

2.6.1.1.4 cursor

GenericConnection.**cursor**(**args*, ***kwargs*) → privex.db.types.GenericCursor

2.6.1.1.5 rollback

GenericConnection.**rollback**(**args*, ***kwargs*)

2.6.2 GenericCursor

class privex.db.types.**GenericCursor**(**args*, ***kwargs*)

This is a `typing_extensions.Protocol` which represents any database Cursor object which follows the Python DB API (PEP 249).

__init__(**args*, ***kwargs*)

2.6.2.1 Methods

Methods

__init__(*args, **kwargs)
close(*args, **kwargs)
execute(query[, params])
executemany(query[, params])
fetchall(*args, **kwargs)
fetchmany(*args, **kwargs)
fetchone(*args, **kwargs)

2.6.2.1.1 __init__

GenericCursor.**__init__**(**args*, ***kwargs*)

2.6.2.1.2 close

GenericCursor.**close**(**args*, ***kwargs*)

2.6.2.1.3 execute

GenericCursor.**execute**(*query: str*, *params: Iterable = None*, **args*, ***kwargs*) → Any

2.6.2.1.4 executemany

GenericCursor.**executemany**(*query: str*, *params: Iterable = None*, **args*, ***kwargs*) → Any

2.6.2.1.5 fetchall

GenericCursor.**fetchall**(**args*, ***kwargs*) → Iterable

2.6.2.1.6 fetchmany

GenericCursor.**fetchmany**(**args*, ***kwargs*) → Iterable

2.6.2.1.7 fetchone

GenericCursor.**fetchone**(**args*, ***kwargs*) → Union[tuple, list, dict, set]

class privex.db.types.**GenericAsyncConnection**(**args*, ***kwargs*)

class privex.db.types.**GenericAsyncCursor**(**args*, ***kwargs*)

class privex.db.types.**GenericConnection**(**args*, ***kwargs*)
This is a `typing_extensions.Protocol` which represents any database Connection object which follows the Python DB API (PEP 249).

class privex.db.types.**GenericCursor**(**args*, ***kwargs*)
This is a `typing_extensions.Protocol` which represents any database Cursor object which follows the Python DB API (PEP 249).

2.7 privex.db.query

2.7.1 privex.db.query.base

Classes

BaseQueryBuilder(table[, connection])	This is an SQL query builder class which outputs ANSI compatible SQL queries, and can use connections/cursors to execute the queries that it builds.
QueryMode	A small `enum.Enum` used for the `query_mode` (whether to return rows as tuples or dicts) with Query Builder classes (see *BaseQueryBuilder* *SqliteQueryBuilder* *PostgresQueryBuilder*)

2.7.1.1 BaseQueryBuilder

class privex.db.query.base.**BaseQueryBuilder**(*table: str*, *connection: privex.db.types.GenericConnection = None*, ***kwargs*)

This is an SQL query builder class which outputs ANSI compatible SQL queries, and can use connections/cursors to execute the queries that it builds.

This is an **abstract base class** (`abc.ABC`) meaning that it's not designed to be constructed directly, instead it should be used as a parent class for a database specific query builder, for example *SqliteQueryBuilder* or *PostgresQueryBuilder*.

To implement a sub-class of *BaseQueryBuilder*, you must:

- Implement all methods marked with `@abstractmethod`, such as *build_query()*, *all()* and *fetch()*
- If your DBMS or it's Python API doesn't follow the default query configuration (see the attributes starting with `Q_`), then you should adjust the `Q_` attributes in your class to match your DB / DB API.

  e.g. Set `Q_DEFAULT_PLACEHOLDER = "?"` if your DB API expects `?` for prepared statement placeholders instead of `%s`.
- While not required, you may wish to implement a constructor (*__init__()*), and override *get_cursor()* to adjust it to your database API requirements

__init__(*table: str*, *connection: privex.db.types.GenericConnection = None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

abstract all(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[Iterable[dict], Iterable[tuple]]

Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = BaseQueryBuilder('people').all()    # Equivalent to ``SELECT *␣
↪FROM people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop. Ideally, should load rows as you iterate, saving RAM.

abstract build_query() → str
Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

abstract fetch(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

abstract fetch_next(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

get_cursor(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*)
Create and return a new database cursor object.

It's recommended to override this method if you're inheriting from this class, as this Generic version of `get_cursor` does not make use of `cursor_name` nor `cursor_class`.

Parameters

- **cursor_name** (*str*) – (If DB API supports it) The name for this cursor
- **cursor_class** (*type*) – (If DB API supports it) The cursor class to use

Return GenericCursor cursor A cursor object which should implement at least the basic Python DB API Cursor functionality as specified in *GenericCursor* ((PEP 249)

group_by(**args*)
Add one or more columns to group by clause.

example: group_by('name', 'date') == GROUP BY name, date

Parameters **args** – One or more columns to group by

Returns QueryBuilder object (for chaining)

limit(*limit_num*, *offset=None*)
Add a limit/offset. When using offset you should use an ORDER BY to avoid issues. :param limit_num: Amount of rows to limit to :param offset: Offset by this many rows (optional) :return: QueryBuilder object (for chaining)

order(**args*, *direction='DESC'*)
example: order('mycol', 'othercol') == ORDER BY mycol, othercol DESC

Parameters

- **args** – One or more order columns as individual args
- **direction** – Direction to sort

Returns QueryBuilder object (for chaining)

order_by(**args*, ***kwargs*)
Alias of *order()*

select (**args*)
Add columns to select clause, specify as individual args. NOTE: no escaping!

example:

q.select('mycol', 'othercol', 'somecol as thiscol')

can also chain: q.select('mycol').select('othercol')

Parameters **args** – columns to select as individual arguments

Returns QueryBuilder object (for chaining)

where (*col*, *val*, *compare='='*, *placeholder=None*)
For adding a simple col=value clause with "AND" before it (if at least 1 other clause). val is escaped properly

example: where('x','test').where('y','thing') produces prepared sql "WHERE x = %s AND y = %s"

Parameters

- **col** – the column, function etc. to query
- **val** – the value it should be equal to. most python objects will be converted and escaped properly
- **compare** – instead of '=', compare using this comparator, e.g. '>', '<=' etc.
- **placeholder** – Set the value placeholder, e.g. placeholder='HOST(%s)'

Returns QueryBuilder object (for chaining)

where_or (*col*, *val*, *compare='='*, *placeholder=None*)
For adding simple col=value clause with "OR" before it (if at least 1 other clause). val is escaped properly

example: where('x','test').where_or('y','thing') produces prepared sql "WHERE x = %s OR y = %s"

Parameters

- **col** – the column, function etc. to query
- **val** – the value it should be equal to. most python objects will be converted and escaped properly
- **compare** – instead of '=', compare using this comparator, e.g. '>', '<=' etc.
- **placeholder** – Set the value placeholder, e.g. placeholder='HOST(%s)'

Returns QueryBuilder object (for chaining)

2.7.1.1.1 Methods

Methods

__init__(table[, connection])	Initialize self.
all([query_mode])	Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)
build_query()	Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

Continued on next page

Table 20 – continued from previous page

close_cursor()	
execute(*args, **kwargs)	
fetch([query_mode])	Executes the current query, and fetches the first result as a `dict`.
fetch_next([query_mode])	Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.
get_cursor([cursor_name, cursor_class])	Create and return a new database cursor object.
group_by(*args)	Add one or more columns to group by clause.
limit(limit_num[, offset])	Add a limit/offset.
order(*args[, direction])	example: order('mycol', 'othercol') == ORDER BY mycol, othercol DESC
order_by(*args, **kwargs)	Alias of *order()*
select(*args)	Add columns to select clause, specify as individual args.
where(col, val[, compare, placeholder])	For adding a simple col=value clause with "AND" before it (if at least 1 other clause).
where_or(col, val[, compare, placeholder])	For adding simple col=value clause with "OR" before it (if at least 1 other clause).

2.7.1.1.1.1 __init__

`BaseQueryBuilder.`**`__init__`**(*table: str*, *connection: privex.db.types.GenericConnection = None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

2.7.1.1.1.2 all

`abstract` `BaseQueryBuilder.`**`all`**(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[Iterable[dict], Iterable[tuple]]

Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = BaseQueryBuilder('people').all()    # Equivalent to ``SELECT * FROM␣
↪people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop. Ideally, should load rows as you iterate, saving RAM.

2.7.1.1.1.3 build_query

abstract BaseQueryBuilder.**build_query**() → str

Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

2.7.1.1.1.4 close_cursor

BaseQueryBuilder.**close_cursor**()

2.7.1.1.1.5 execute

BaseQueryBuilder.**execute**(**args*, ***kwargs*)

2.7.1.1.1.6 fetch

abstract BaseQueryBuilder.**fetch**(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]

Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

2.7.1.1.1.7 fetch_next

abstract BaseQueryBuilder.**fetch_next**(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]

Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

2.7.1.1.1.8 get_cursor

BaseQueryBuilder.**get_cursor**(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*)

Create and return a new database cursor object.

It's recommended to override this method if you're inheriting from this class, as this Generic version of `get_cursor` does not make use of `cursor_name` nor `cursor_class`.

Parameters

- **cursor_name** (*str*) – (If DB API supports it) The name for this cursor
- **cursor_class** (*type*) – (If DB API supports it) The cursor class to use

Return GenericCursor cursor A cursor object which should implement at least the basic Python DB API Cursor functionality as specified in *GenericCursor* ((PEP 249)

2.7.1.1.1.9 group_by

BaseQueryBuilder.**group_by**(**args*)

Add one or more columns to group by clause.

example: group_by('name', 'date') == GROUP BY name, date

Parameters **args** – One or more columns to group by

Returns QueryBuilder object (for chaining)

2.7.1.1.1.10 limit

BaseQueryBuilder.**limit**(*limit_num*, *offset=None*)

Add a limit/offset. When using offset you should use an ORDER BY to avoid issues. :param limit_num: Amount of rows to limit to :param offset: Offset by this many rows (optional) :return: QueryBuilder object (for chaining)

2.7.1.1.1.11 order

BaseQueryBuilder.**order**(**args*, *direction='DESC'*)

example: order('mycol', 'othercol') == ORDER BY mycol, othercol DESC

Parameters

- **args** – One or more order columns as individual args
- **direction** – Direction to sort

Returns QueryBuilder object (for chaining)

2.7.1.1.1.12 order_by

BaseQueryBuilder.**order_by**(**args*, ***kwargs*)

Alias of *order()*

2.7.1.1.1.13 select

BaseQueryBuilder.**select**(**args*)

Add columns to select clause, specify as individual args. NOTE: no escaping!

example:

q.select('mycol', 'othercol', 'somecol as thiscol')

can also chain: q.select('mycol').select('othercol')

Parameters **args** – columns to select as individual arguments

Returns QueryBuilder object (for chaining)

2.7.1.1.1.14 where

BaseQueryBuilder.**where**(*col*, *val*, *compare='='*, *placeholder=None*)

For adding a simple col=value clause with "AND" before it (if at least 1 other clause). val is escaped properly

example: where('x','test').where('y','thing') produces prepared sql "WHERE x = %s AND y = %s"

Parameters

- **col** – the column, function etc. to query
- **val** – the value it should be equal to. most python objects will be converted and escaped properly
- **compare** – instead of '=', compare using this comparator, e.g. '>', '<=' etc.
- **placeholder** – Set the value placeholder, e.g. placeholder='HOST(%s)'

Returns QueryBuilder object (for chaining)

2.7.1.1.1.15 where_or

BaseQueryBuilder.**where_or**(*col*, *val*, *compare='='*, *placeholder=None*)

For adding simple col=value clause with "OR" before it (if at least 1 other clause). val is escaped properly

example: where('x','test').where_or('y','thing') produces prepared sql "WHERE x = %s OR y = %s"

Parameters

- **col** – the column, function etc. to query
- **val** – the value it should be equal to. most python objects will be converted and escaped properly
- **compare** – instead of '=', compare using this comparator, e.g. '>', '<=' etc.
- **placeholder** – Set the value placeholder, e.g. placeholder='HOST(%s)'

Returns QueryBuilder object (for chaining)

2.7.1.1.2 Attributes

Attributes

Q_DEFAULT_PLACEHOLDER
Q_GROUP_BY_CLAUSE
Q_LIMIT_CLAUSE
Q_OFFSET_CLAUSE
Q_ORDER_CLAUSE
Q_POST_QUERY
Q_PRE_QUERY
Q_SELECT_CLAUSE
Q_WHERE_CLAUSE
connection
cursor

2.7.1.1.2.1 Q_DEFAULT_PLACEHOLDER

BaseQueryBuilder.**Q_DEFAULT_PLACEHOLDER = '%s'**

2.7.1.1.2.2 Q_GROUP_BY_CLAUSE

BaseQueryBuilder.**Q_GROUP_BY_CLAUSE = ' GROUP BY {group_cols}'**

2.7.1.1.2.3 Q_LIMIT_CLAUSE

BaseQueryBuilder.**Q_LIMIT_CLAUSE = ' LIMIT {limit}'**

2.7.1.1.2.4 Q_OFFSET_CLAUSE

BaseQueryBuilder.**Q_OFFSET_CLAUSE = ' OFFSET {offset}'**

2.7.1.1.2.5 Q_ORDER_CLAUSE

BaseQueryBuilder.**Q_ORDER_CLAUSE = ' ORDER BY {order_cols} {order_dir}'**

2.7.1.1.2.6 Q_POST_QUERY

BaseQueryBuilder.**Q_POST_QUERY = ''**

2.7.1.1.2.7 Q_PRE_QUERY

BaseQueryBuilder.**Q_PRE_QUERY = ''**

2.7.1.1.2.8 Q_SELECT_CLAUSE

BaseQueryBuilder.**Q_SELECT_CLAUSE = ' SELECT {cols} FROM {table}'**

2.7.1.1.2.9 Q_WHERE_CLAUSE

BaseQueryBuilder.**Q_WHERE_CLAUSE = ' WHERE {w_clauses}'**

2.7.1.1.2.10 connection

BaseQueryBuilder.**connection = None**

2.7.1.1.2.11 cursor

property BaseQueryBuilder.**cursor**

2.7.1.2 QueryMode

class privex.db.query.base.**QueryMode**

A small enum.Enum used for the query_mode (whether to return rows as tuples or dicts) with Query Builder classes (see *BaseQueryBuilder* *SqliteQueryBuilder* *PostgresQueryBuilder*)

__init__()

Initialize self. See help(type(self)) for accurate signature.

2.7.1.2.1 Attributes

Attributes

DEFAULT
ROW_DICT
ROW_TUPLE

2.7.1.2.1.1 DEFAULT

QueryMode.**DEFAULT = 'default'**

2.7.1.2.1.2 ROW_DICT

QueryMode.**ROW_DICT = 'dict'**

2.7.1.2.1.3 ROW_TUPLE

QueryMode.**ROW_TUPLE = 'tuple'**

class privex.db.query.base.**BaseQueryBuilder**(*table: str*, *connection: privex.db.types.GenericConnection = None*, ***kwargs*)

This is an SQL query builder class which outputs ANSI compatible SQL queries, and can use connections/cursors to execute the queries that it builds.

This is an **abstract base class** (abc.ABC) meaning that it's not designed to be constructed directly, instead it should be used as a parent class for a database specific query builder, for example *SqliteQueryBuilder* or *PostgresQueryBuilder*.

To implement a sub-class of *BaseQueryBuilder*, you must:

- Implement all methods marked with `@abstractmethod`, such as *build_query()*, *all()* and *fetch()*
- If your DBMS or it's Python API doesn't follow the default query configuration (see the attributes starting with `Q_`), then you should adjust the `Q_` attributes in your class to match your DB / DB API.

  e.g. Set `Q_DEFAULT_PLACEHOLDER = "?"` if your DB API expects `?` for prepared statement placeholders instead of `%s`.
- While not required, you may wish to implement a constructor (*__init__()*), and override *get_cursor()* to adjust it to your database API requirements

abstract all(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[Iterable[dict], Iterable[tuple]]

Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = BaseQueryBuilder('people').all()    # Equivalent to ``SELECT *␣
↪FROM people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop. Ideally, should load rows as you iterate, saving RAM.

abstract build_query() → str

Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

abstract fetch(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]

Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

abstract fetch_next(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]

Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

get_cursor(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*)

Create and return a new database cursor object.

It's recommended to override this method if you're inheriting from this class, as this Generic version of `get_cursor` does not make use of `cursor_name` nor `cursor_class`.

Parameters

- **cursor_name** (*str*) – (If DB API supports it) The name for this cursor
- **cursor_class** (*type*) – (If DB API supports it) The cursor class to use

Return GenericCursor cursor A cursor object which should implement at least the basic Python DB API Cursor functionality as specified in *GenericCursor* ((PEP 249)

group_by(**args*)
Add one or more columns to group by clause.

example: group_by('name', 'date') == GROUP BY name, date

Parameters **args** – One or more columns to group by

Returns QueryBuilder object (for chaining)

limit(*limit_num*, *offset=None*)
Add a limit/offset. When using offset you should use an ORDER BY to avoid issues. :param limit_num: Amount of rows to limit to :param offset: Offset by this many rows (optional) :return: QueryBuilder object (for chaining)

order(**args*, *direction='DESC'*)
example: order('mycol', 'othercol') == ORDER BY mycol, othercol DESC

Parameters

- **args** – One or more order columns as individual args
- **direction** – Direction to sort

Returns QueryBuilder object (for chaining)

order_by(**args*, ***kwargs*)
Alias of *order()*

select(**args*)
Add columns to select clause, specify as individual args. NOTE: no escaping!

example:

q.select('mycol', 'othercol', 'somecol as thiscol')

can also chain: q.select('mycol').select('othercol')

Parameters **args** – columns to select as individual arguments

Returns QueryBuilder object (for chaining)

where(*col*, *val*, *compare='='*, *placeholder=None*)
For adding a simple col=value clause with "AND" before it (if at least 1 other clause). val is escaped properly

example: where('x','test').where('y','thing') produces prepared sql "WHERE x = %s AND y = %s"

Parameters

- **col** – the column, function etc. to query
- **val** – the value it should be equal to. most python objects will be converted and escaped properly
- **compare** – instead of '=', compare using this comparator, e.g. '>', '<=' etc.
- **placeholder** – Set the value placeholder, e.g. placeholder='HOST(%s)'

Returns QueryBuilder object (for chaining)

where_or(*col*, *val*, *compare='='*, *placeholder=None*)
For adding simple col=value clause with "OR" before it (if at least 1 other clause). val is escaped properly

example: where('x','test').where_or('y','thing') produces prepared sql "WHERE x = %s OR y = %s"

Parameters

- **col** – the column, function etc. to query

- **val** – the value it should be equal to. most python objects will be converted and escaped properly
- **compare** – instead of '=', compare using this comparator, e.g. '>', '<=' etc.
- **placeholder** – Set the value placeholder, e.g. placeholder='HOST(%s)'

Returns QueryBuilder object (for chaining)

class privex.db.query.base.**QueryMode**

A small `enum.Enum` used for the `query_mode` (whether to return rows as tuples or dicts) with Query Builder classes (see *BaseQueryBuilder SqliteQueryBuilder PostgresQueryBuilder*)

2.7.2 privex.db.query.postgres

Classes

PostgresQueryBuilder(table[, connection])	A simple SQL query builder / ORM, designed for use with PostgreSQL.

2.7.2.1 PostgresQueryBuilder

class privex.db.query.postgres.**PostgresQueryBuilder**(*table: str*, *connection=None*, ***kwargs*)

A simple SQL query builder / ORM, designed for use with PostgreSQL. May or may not work with other RDBMS's.

Basic Usage:

First, inject your psycopg2 connection into QueryBuilder, so it's available to all instances.

```
>>> PostgresQueryBuilder.conn = psycopg2.connect(user='bob', dbname='my_db
↪')
```

Now, just construct the class, passing the table name to query.

```
>>> q = PostgresQueryBuilder('orders')
```

You can execute each query building method either on their own line, and/or you can chain them together.

WARNING: many methods such as *select()* do not escape your input. Only *where()* and *where_or()* use prepared statements, with a placeholder for the value you pass.

```
>>> q.select('full_name', 'address')
>>> q.select('SUM(order_amt) as total_spend').where('country', 'FR')
↪   ...      .where('SUM(order_amt)', '100', compare='>=')
>>> q.group_by('full_name', 'address')
```

Once you've finished building your query, simply call either *all()* (return all results as a list) or *fetch()* (returns the first result, or `None` if there's no match)

```
>>> results = q.order('full_name', direction='ASC').all()
>>> print(results[0])
```

Output:

```
dict{'full_name': 'Aaron Doe', 'address': '123 Fake St.', 'total_spend':␣
↪127.88}
```

You can call *build_query()* to see the query that would be sent to PostgreSQL, showing the value placeholders (e.g. %s)

```
>>> print(q.build_query())
```

Output:

```
SELECT full_name, address, SUM(order_amt) as total_spend FROM orders␣
↪WHERE country = %s
AND SUM(order_amt) >= %s GROUP BY full_name, address ORDER BY full_name␣
↪ASC;
```

Copyright:

```
+==================================================+
|                 © 2019 Privex Inc.               |
|               https://www.privex.io              |
+==================================================+
|                                                  |
|        Privex Database Library                   |
|                                                  |
|        Core Developer(s):                        |
|                                                  |
|          (+)  Chris (@someguy123) [Privex]       |
|                                                  |
+==================================================+
```

__init__(*table: str*, *connection=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

all(*query_mode=<QueryMode.DEFAULT: 'default'>*) → Union[Iterable[dict], Iterable[tuple]]
Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = PostgresQueryBuilder('people').all()    # Equivalent to ``SELECT␣
↪* FROM people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop, loads rows as you iterate, saving RAM

build_query() → str
Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

fetch(*query_mode=<QueryMode.DEFAULT: 'default'>*) → Union[dict, tuple, None]
Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

fetch_next(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

get_cursor(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*) → psycopg2.extensions.cursor
Create and return a new Postgres cursor object

query_mode_cursor(*query_mode: privex.db.query.base.QueryMode*, *replace_cursor=True*, *cursor_mgr=True*)
Return a cursor object with the cursor class based on the `query_mode`, using the query_mode to cursor class map in `_cursor_map`

Parameters

- **query_mode** (QueryMode) – The QueryMode to obtain a cursor for
- **replace_cursor** (*bool*) – (Default: `True`) If True, replace the shared instance `_cursor` with this new cursor.
- **cursor_mgr** (*bool*) – Wrap the cursor object in *CursorManager*

Returns

select_date(**args*)
Add columns to be returned as an ISO formatted date to the select clause. Specify as individual args. Do not use 'col AS x'. NOTE: no escaping is used!

example: q.select_date('created_at', 'updated_at') can also chain: q.select_date('mycol').select_date('othercol')

Parameters **args** – date columns to select as individual arguments

Returns QueryBuilder object (for chaining)

2.7.2.1.1 Methods

Methods

__init__(table[, connection])	Initialize self.
all([query_mode])	Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)
build_query()	Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string
fetch([query_mode])	Executes the current query, and fetches the first result as a `dict`.
fetch_next([query_mode])	Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.
get_cursor([cursor_name, cursor_class])	Create and return a new Postgres cursor object

Continued on next page

Table 24 – continued from previous page

query_mode_cursor(query_mode[, ...])	Return a cursor object with the cursor class based on the `query_mode`, using the query_mode to cursor class map in `_cursor_map`
select_date(*args)	Add columns to be returned as an ISO formatted date to the select clause.

2.7.2.1.1.1 __init__

`PostgresQueryBuilder.`**`__init__`**(*table: str*, *connection=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

2.7.2.1.1.2 all

`PostgresQueryBuilder.`**`all`**(*query_mode=<QueryMode.DEFAULT: 'default'>*) → Union[Iterable[dict], Iterable[tuple]]

Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = PostgresQueryBuilder('people').all()    # Equivalent to ``SELECT *␣
↪FROM people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop, loads rows as you iterate, saving RAM

2.7.2.1.1.3 build_query

`PostgresQueryBuilder.`**`build_query`**`()` → str

Used internally by *`all()`* and *`fetch()`* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

2.7.2.1.1.4 fetch

`PostgresQueryBuilder.`**`fetch`**(*query_mode=<QueryMode.DEFAULT: 'default'>*) → Union[dict, tuple, None]

Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

2.7.2.1.1.5 fetch_next

PostgresQueryBuilder.**fetch_next**(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]

Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

2.7.2.1.1.6 get_cursor

PostgresQueryBuilder.**get_cursor**(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*) → psycopg2.extensions.cursor

Create and return a new Postgres cursor object

2.7.2.1.1.7 query_mode_cursor

PostgresQueryBuilder.**query_mode_cursor**(*query_mode: privex.db.query.base.QueryMode*, *replace_cursor=True*, *cursor_mgr=True*)

Return a cursor object with the cursor class based on the `query_mode`, using the query_mode to cursor class map in `_cursor_map`

Parameters

- **query_mode** (QueryMode) – The QueryMode to obtain a cursor for
- **replace_cursor** (*bool*) – (Default: `True`) If True, replace the shared instance `_cursor` with this new cursor.
- **cursor_mgr** (*bool*) – Wrap the cursor object in *CursorManager*

Returns

2.7.2.1.1.8 select_date

PostgresQueryBuilder.**select_date**(**args*)

Add columns to be returned as an ISO formatted date to the select clause. Specify as individual args. Do not use 'col AS x'. NOTE: no escaping is used!

example: q.select_date('created_at', 'updated_at') can also chain: q.select_date('mycol').select_date('othercol')

Parameters **args** – date columns to select as individual arguments

Returns QueryBuilder object (for chaining)

2.7.2.1.2 Attributes

Attributes

Q_DEFAULT_PLACEHOLDER
Q_PRE_QUERY
conn
cursor

2.7.2.1.2.1 Q_DEFAULT_PLACEHOLDER

PostgresQueryBuilder.**Q_DEFAULT_PLACEHOLDER = '%s'**

2.7.2.1.2.2 Q_PRE_QUERY

PostgresQueryBuilder.**Q_PRE_QUERY = "set timezone to 'UTC'; "**

2.7.2.1.2.3 conn

property PostgresQueryBuilder.**conn**

2.7.2.1.2.4 cursor

property PostgresQueryBuilder.**cursor**

class privex.db.query.postgres.**PostgresQueryBuilder**(*table: str*, *connection=None*, ***kwargs*)

A simple SQL query builder / ORM, designed for use with PostgreSQL. May or may not work with other RDBMS's.

Basic Usage:

First, inject your psycopg2 connection into QueryBuilder, so it's available to all instances.

```
>>> PostgresQueryBuilder.conn = psycopg2.connect(user='bob', dbname='my_db
↪')
```

Now, just construct the class, passing the table name to query.

```
>>> q = PostgresQueryBuilder('orders')
```

You can execute each query building method either on their own line, and/or you can chain them together.

WARNING: many methods such as *select()* do not escape your input. Only *where()* and *where_or()* use prepared statements, with a placeholder for the value you pass.

```
>>> q.select('full_name', 'address')
>>> q.select('SUM(order_amt) as total_spend').where('country', 'FR')
↪   ...     .where('SUM(order_amt)', '100', compare='>=')
>>> q.group_by('full_name', 'address')
```

Once you've finished building your query, simply call either *all()* (return all results as a list) or *fetch()* (returns the first result, or `None` if there's no match)

```
>>> results = q.order('full_name', direction='ASC').all()
>>> print(results[0])
```

Output:

```
dict{'full_name': 'Aaron Doe', 'address': '123 Fake St.', 'total_spend':␣
↪127.88}
```

You can call *build_query()* to see the query that would be sent to PostgreSQL, showing the value placeholders (e.g. %s)

```
>>> print(q.build_query())
```

Output:

```
SELECT full_name, address, SUM(order_amt) as total_spend FROM orders␣
↪WHERE country = %s
AND SUM(order_amt) >= %s GROUP BY full_name, address ORDER BY full_name␣
↪ASC;
```

Copyright:

```
+==================================================+
|                 © 2019 Privex Inc.               |
|               https://www.privex.io              |
+==================================================+
|                                                  |
|        Privex Database Library                   |
|                                                  |
|        Core Developer(s):                        |
|                                                  |
|          (+)  Chris (@someguy123) [Privex]       |
|                                                  |
+==================================================+
```

all(*query_mode=<QueryMode.DEFAULT: 'default'>*) → Union[Iterable[dict], Iterable[tuple]]
Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = PostgresQueryBuilder('people').all()    # Equivalent to ``SELECT␣
↪* FROM people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop, loads rows as you iterate, saving RAM

build_query() → str
Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

fetch(*query_mode=<QueryMode.DEFAULT: 'default'>*) → Union[dict, tuple, None]
Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

fetch_next(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

get_cursor(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*) → psycopg2.extensions.cursor
Create and return a new Postgres cursor object

query_mode_cursor(*query_mode: privex.db.query.base.QueryMode*, *replace_cursor=True*, *cursor_mgr=True*)
Return a cursor object with the cursor class based on the `query_mode`, using the query_mode to cursor class map in `_cursor_map`

Parameters

- **query_mode** (QueryMode) – The QueryMode to obtain a cursor for
- **replace_cursor** (*bool*) – (Default: `True`) If True, replace the shared instance `_cursor` with this new cursor.
- **cursor_mgr** (*bool*) – Wrap the cursor object in *CursorManager*

Returns

select_date(**args*)
Add columns to be returned as an ISO formatted date to the select clause. Specify as individual args. Do not use 'col AS x'. NOTE: no escaping is used!

example: q.select_date('created_at', 'updated_at') can also chain: q.select_date('mycol').select_date('othercol')

Parameters **args** – date columns to select as individual arguments

Returns QueryBuilder object (for chaining)

2.7.3 privex.db.query.sqlite

Classes

SqliteQueryBuilder(table[, connection])

2.7.3.1 SqliteQueryBuilder

class privex.db.query.sqlite.**SqliteQueryBuilder**(*table: str*, *connection: privex.db.types.GenericConnection = None*, ***kwargs*)

__init__(*table: str*, *connection: privex.db.types.GenericConnection = None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

all(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[Iterable[dict], Iterable[tuple]]
Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = BaseQueryBuilder('people').all()    # Equivalent to ``SELECT *␣
↪FROM people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop. Ideally, should load rows as you iterate, saving RAM.

build_query() → str
Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

fetch(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

fetch_next(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

2.7.3.1.1 Methods

Methods

all([query_mode])	Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)
build_query()	Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

Continued on next page

Table 27 – continued from previous page

fetch([query_mode])	Executes the current query, and fetches the first result as a `dict`.
fetch_next([query_mode])	Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

2.7.3.1.1.1 all

`SqliteQueryBuilder.`**`all`**(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[Iterable[dict], Iterable[tuple]]

Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = BaseQueryBuilder('people').all()    # Equivalent to ``SELECT * FROM␣
↪people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop. Ideally, should load rows as you iterate, saving RAM.

2.7.3.1.1.2 build_query

`SqliteQueryBuilder.`**`build_query`**`()` → str

Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

2.7.3.1.1.3 fetch

`SqliteQueryBuilder.`**`fetch`**(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]

Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

2.7.3.1.1.4 fetch_next

`SqliteQueryBuilder.`**`fetch_next`**(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]

Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **`query_mode`** (`QueryMode`) –

Returns

2.7.3.1.2 Attributes

Attributes

Q_DEFAULT_PLACEHOLDER
Q_PRE_QUERY
conn
connection

2.7.3.1.2.1 Q_DEFAULT_PLACEHOLDER

SqliteQueryBuilder.**Q_DEFAULT_PLACEHOLDER = '?'**

2.7.3.1.2.2 Q_PRE_QUERY

SqliteQueryBuilder.**Q_PRE_QUERY = ''**

2.7.3.1.2.3 conn

property SqliteQueryBuilder.**conn**

2.7.3.1.2.4 connection

SqliteQueryBuilder.**connection: sqlite3.Connection = None**

class privex.db.query.sqlite.**SqliteQueryBuilder**(*table: str, connection: privex.db.types.GenericConnection = None, **kwargs*)

all(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[Iterable[dict], Iterable[tuple]]
Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = BaseQueryBuilder('people').all()    # Equivalent to ``SELECT *␣
↪FROM people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop. Ideally, should load rows as you iterate, saving RAM.

build_query() → str
Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

fetch(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

fetch_next(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

2.8 How to use the unit tests

This module contains test cases for Privex's Python Database Wrappers (privex-db).

2.8.1 Testing pre-requisites

- Ensure you have any mandatory requirements installed (see setup.py's install_requires)
- You should install `pytest` to run the tests, it works much better than standard python unittest.
- You may wish to install any optional requirements listed in README.md for best results
- Python 3.7 is recommended at the time of writing this. See README.md in-case this has changed.

For the best testing experience, it's recommended to install the `dev` extra, which includes every optional dependency, as well as development requirements such as `pytest` , `coverage` as well as requirements for building the documentation.

2.8.2 Running via PyTest

To run the tests, we strongly recommend using the `pytest` tool (used by default for our Travis CI):

```
# Install PyTest if you don't already have it.
user@host: ~/privex-db $ pip3 install pytest

# We recommend adding the option ``-rxXs`` which will show information about why␣
↪certain tests were skipped
# as well as info on xpass / xfail tests
# You can add `-v` for more detailed output, just like when running the tests␣
↪directly.
user@host: ~/privex-db $ pytest -rxXs

# NOTE: If you're using a virtualenv, sometimes you may encounter strange conflicts␣
↪between a global install
# of PyTest, and the virtualenv PyTest, resulting in errors related to packages not␣
↪being installed.
# A simple workaround is just to call pytest as a module from the python3 executable:

user@host: ~/privex-db $ python3 -m pytest -rxXs


============================================= test session starts␣
↪=============================================
platform darwin -- Python 3.8.0, pytest-5.3.1, py-1.8.0, pluggy-0.13.1
```

(continues on next page)

(continued from previous page)

```
cachedir: .pytest_cache
rootdir: /home/user/privex-db, inifile: pytest.ini
plugins: cov-2.8.1
collected 23 items

tests/test_postgres_builder.py::TestPostgresBuilder::test_all_call SKIPPED
↪                       [  4%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_group_call SKIPPED
↪                       [  8%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_query_all SKIPPED
↪                       [ 13%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_where_call SKIPPED
↪                       [ 34%]
tests/test_sqlite_builder.py::TestSQLiteBuilder::test_all_call PASSED
↪                       [ 39%]
tests/test_sqlite_builder.py::TestSQLiteBuilder::test_group_call PASSED
↪                       [ 43%]
tests/test_sqlite_builder.py::TestSQLiteBuilder::test_query_select_col_where_group␣
↪PASSED                   [ 56%]
tests/test_sqlite_builder.py::TestSQLiteBuilder::test_query_where_first_name_last_
↪name PASSED               [ 65%]
tests/test_sqlite_builder.py::TestSQLiteBuilder::test_where_call PASSED
↪                       [ 69%]
tests/test_sqlite_wrapper.py::TestSQLiteWrapper::test_find_user_dict_mode PASSED
↪                       [ 73%]
tests/test_sqlite_wrapper.py::TestSQLiteWrapper::test_insert_find_user PASSED
↪                       [ 91%]
tests/test_sqlite_wrapper.py::TestSQLiteWrapper::test_tables_created PASSED
↪                       [ 95%]
tests/test_sqlite_wrapper.py::TestSQLiteWrapper::test_tables_drop PASSED
↪                       [100%]

=============================================== short test summary info␣
↪================================================
SKIPPED [1] tests/test_postgres_builder.py:132: Library 'psycopg2' is not installed...
SKIPPED [1] tests/test_postgres_builder.py:159: Library 'psycopg2' is not installed...
SKIPPED [1] tests/test_postgres_builder.py:78: Library 'psycopg2' is not installed...
SKIPPED [1] tests/test_postgres_builder.py:91: Library 'psycopg2' is not installed...
SKIPPED [1] tests/test_postgres_builder.py:116: Library 'psycopg2' is not installed...
SKIPPED [1] tests/test_postgres_builder.py:102: Library 'psycopg2' is not installed...
SKIPPED [1] tests/test_postgres_builder.py:84: Library 'psycopg2' is not installed...
SKIPPED [1] tests/test_postgres_builder.py:146: Library 'psycopg2' is not installed...
========================================== 15 passed, 8 skipped in 0.13s␣
↪==========================================
```

2.8.3 Running individual test modules

Sometimes, you just want to run only a specific test file.

Thankfully, PyTest allows you to run individual test modules like this:

```
user@host: ~/privex-db $ pytest -rxXs -v tests/test_postgres_builder.py
============================================== test session starts␣
↪==============================================
platform darwin -- Python 3.8.0, pytest-5.3.1, py-1.8.0, pluggy-0.13.1
```

(continues on next page)

(continued from previous page)

```
cachedir: .pytest_cache
rootdir: /home/user/privex-db, inifile: pytest.ini
plugins: cov-2.8.1
collected 8 items

tests/test_postgres_builder.py::TestPostgresBuilder::test_all_call PASSED
↪                       [ 12%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_group_call PASSED
↪                       [ 25%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_query_all PASSED
↪                       [ 37%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_query_select_col_where
↪PASSED                       [ 50%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_query_select_col_where_
↪group PASSED                 [ 62%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_query_select_col_where_
↪order PASSED                 [ 75%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_query_where_first_name_last_
↪name PASSED             [ 87%]
tests/test_postgres_builder.py::TestPostgresBuilder::test_where_call PASSED
↪                       [100%]

============================================== 8 passed in 0.17s
↪==============================================
```

Copyright:

```
    +==================================================+
    |                © 2019 Privex Inc.                |
    |              https://www.privex.io               |
    +==================================================+
    |                                                  |
    |        Privex's Python Database Library          |
    |        License: X11 / MIT                        |
    |                                                  |
    |        Originally Developed by Privex Inc.       |
    |        Core Developer(s):                        |
    |                                                  |
    |          (+)  Chris (@someguy123) [Privex]       |
    |                                                  |
    +==================================================+

Copyright 2019     Privex Inc.    ( https://www.privex.io )
```

2.9 Unit Test List / Overview

base	This file contains code shared between tests, such as *PrivexDBTestBase* which is the base class shared by all unit tests.
test_postgres	Tests related to *PostgresQueryBuilder*, *ExamplePostgresWrapper* and *PostgresWrapper*

Continued on next page

Table 29 – continued from previous page

test_sqlite_builder	Tests related to *SqliteQueryBuilder* and *ExampleWrapper*
test_sqlite_wrapper	Tests related to *SqliteWrapper* / *ExampleWrapper*

2.9.1 tests.base

This file contains code shared between tests, such as *PrivexDBTestBase* which is the base class shared by all unit tests.

Copyright:

```
+===================================================+
|                 © 2019 Privex Inc.                |
|               https://www.privex.io               |
+===================================================+
|                                                   |
|        Django Database Lock Manager               |
|        License: X11/MIT                           |
|                                                   |
|        Core Developer(s):                         |
|                                                   |
|          (+)  Chris (@someguy123) [Privex]        |
|                                                   |
+===================================================+
```

Classes

ExampleWrapper(*args, **kwargs)	
PrivexDBTestBase([methodName])	Base class for all privex-db test classes.
User(*args, **kwargs)	

2.9.1.1 ExampleWrapper

class tests.base.**ExampleWrapper**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)

Parameters

- **db** (*str*) – Relative / absolute path to SQLite3 database file to use.
- **isolation_level** – Isolation level for SQLite3 connection. Defaults to `None` (auto-commit). See the Python SQLite3 Docs for more information.

Key int db_timeout Amount of time to wait for any SQLite3 locks to expire before giving up

Key str query_mode Either `'flat'` (query returns tuples) or `'dict'` (query returns dicts). More details in PyDoc block under *query_mode*

Key bool memory_persist Use a shared in-memory database, which can be accessed by other instances of this class (in this process) - which is cleared after all memory connections are closed. Shortcut for `db='file::memory:?cache=shared'`

2.9.1.1.1 Methods

Methods

__init__(*args, **kwargs)	**param str db** Relative / absolute path to SQLite3 database file to use.

2.9.1.1.1.1 __init__

ExampleWrapper.**__init__**(**args*, ***kwargs*)

Parameters

- **db** (*str*) – Relative / absolute path to SQLite3 database file to use.
- **isolation_level** – Isolation level for SQLite3 connection. Defaults to `None` (autocommit). See the Python SQLite3 Docs for more information.

Key int db_timeout Amount of time to wait for any SQLite3 locks to expire before giving up

Key str query_mode Either `'flat'` (query returns tuples) or `'dict'` (query returns dicts). More details in PyDoc block under *query_mode*

Key bool memory_persist Use a shared in-memory database, which can be accessed by other instances of this class (in this process) - which is cleared after all memory connections are closed. Shortcut for `db='file::memory:?cache=shared'`

2.9.1.1.2 Attributes

Attributes

DEFAULT_DB
SCHEMAS

2.9.1.1.2.1 DEFAULT_DB

ExampleWrapper.**DEFAULT_DB: str = ':memory:'**

2.9.1.1.2.2 SCHEMAS

ExampleWrapper.**SCHEMAS: List[Tuple[str, str]] = [('users', 'CREATE TABLE users (id INTEGER**

2.9.1.2 PrivexDBTestBase

class tests.base.**PrivexDBTestBase**(*methodName='runTest'*)

Base class for all privex-db test classes. Includes *tearDown()* to reset database after each test.

__init__(*methodName='runTest'*)

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

setUp() → None

Hook method for setting up the test fixture before exercising it.

tearDown() → None

Hook method for deconstructing the test fixture after testing it.

2.9.1.2.1 Methods

Methods

setUp()	Hook method for setting up the test fixture before exercising it.
tearDown()	Hook method for deconstructing the test fixture after testing it.

2.9.1.2.1.1 setUp

PrivexDBTestBase.**setUp**() → None

Hook method for setting up the test fixture before exercising it.

2.9.1.2.1.2 tearDown

PrivexDBTestBase.**tearDown**() → None

Hook method for deconstructing the test fixture after testing it.

2.9.1.2.2 Attributes

Attributes

—

2.9.1.3 User

class tests.base.**User**(**args*, ***kwargs*)

__init__(**args*, ***kwargs*)

2.9.1.3.1 Methods

Methods

—

2.9.1.3.2 Attributes

Attributes

—

class tests.base.**PrivexDBTestBase**(*methodName='runTest'*)

Base class for all privex-db test classes. Includes *tearDown()* to reset database after each test.

setUp() → None

Hook method for setting up the test fixture before exercising it.

tearDown() → None

Hook method for deconstructing the test fixture after testing it.

class tests.base.**SqliteWrapper**(*db: str = None*, *isolation_level=None*, ***kwargs*)

Lightweight wrapper class for interacting with Sqlite3 databases.

Simple direct class usage

```
>>> db_path = expanduser('~/.my_app/my_db.db')
>>> db = SqliteWrapper(db=db_path)
>>> users = db.fetchall("SELECT * FROM users;")
```

Usage

Below is an example wrapper class which uses *SqliteWrapper* as it's parent class.

It overrides the class attributes *DEFAULT_DB_FOLDER*, *DEFAULT_DB_NAME*, and *DEFAULT_DB* - so that if no database path is passed to `MyManager`, then the database file path contained in `MyManager.DEFAULT_DB` will be used as a default.

It also overrides *SCHEMAS* to define 2 tables (`users` and `items`) which will be automatically created when the class is instantiated, unless they already exist.

It adds two methods `get_items` (returns an iterator

```
class MyManager(SqliteWrapper):
    ###
    # If a database path isn't specified, then the class attribute DEFAULT_DB␣
↪will be used.
    ###
    DEFAULT_DB_FOLDER: str = expanduser('~/.my_app')
    DEFAULT_DB_NAME: str = 'my_app.db'
```

(continues on next page)

(continued from previous page)

```
    DEFAULT_DB: str = join(DEFAULT_DB_FOLDER, DEFAULT_DB_NAME)

    ###
    # The SCHEMAS class attribute contains a list of tuples, with each tuple␣
↪containing the name of a
    # table, as well as the SQL query required to create the table if it doesn't␣
↪exist.
    ###
    SCHEMAS: List[Tuple[str, str]] = [
        ('users', "CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"),
        ('items', "CREATE TABLE items (id INTEGER PRIMARY KEY AUTOINCREMENT, name␣
↪TEXT);"),
    ]

    def get_items(self):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchall with a pre-defined SQL query
        return self.fetchall("SELECT * FROM items;")

    def find_item(self, id: int):
        # This is an example of a helper method you might want to define, which␣
↪simply calls
        # self.fetchone with a pre-defined SQL query, and interpolates the 'id'␣
↪parameter into
        # the prepared statement.
        return self.fetchone("SELECT * FROM items WHERE id = ?;", [id]);
```

DEFAULT_DB: str = '/home/docs/.privex_sqlite/privex_sqlite.db'
Combined *DEFAULT_DB_FOLDER* and *DEFAULT_DB_NAME* used as default absolute path for the sqlite3 database

DEFAULT_DB_FOLDER: str = '/home/docs/.privex_sqlite'
If an absolute path isn't given, store the sqlite3 database file in this folder

DEFAULT_DB_NAME: str = 'privex_sqlite.db'
If no database is specified to *__init__()*, then use this (appended to *DEFAULT_DB_FOLDER*)

property conn
Get or create an SQLite3 connection using DB file *db* and return it

db: str = None
Path to the SQLite3 database for this class instance

insert(*_table: str*, *_cursor: sqlite3.Cursor = None*, ***fields*) → Union[privex.helpers.collections.DictObject, sqlite3.Cursor]
Builds and executes an insert query into the table `_table` using the keyword arguments for column names and values.

```
>>> db = GenericDBWrapper(db='SomeDB')
>>> cur = db.insert('users', first_name='John', last_name='Doe', phone='+1-
↪800-123-4567')
>>> cur.lastrowid
15
```

Parameters

- **_table** (*str*) – The table to insert into
- **_cursor** (GenericCursor) – Optionally, specify a cursor to use, instead of the default *cursor*
- **fields** – Keyword args mapping column names to values

Return DictObject cur If no custom cursor was specified, the cursor used to execute the query is converted into a `DictObject` before closing it, then the dict is returned.

Return GenericCursor cur If a custom cursor (`_cursor`) was specified, then that cursor will NOT be auto-closed, and the original cursor instance will be returned.

class tests.base.**BaseQueryBuilder**(*table: str*, *connection: privex.db.types.GenericConnection = None*, ***kwargs*)

This is an SQL query builder class which outputs ANSI compatible SQL queries, and can use connections/cursors to execute the queries that it builds.

This is an **abstract base class** (`abc.ABC`) meaning that it's not designed to be constructed directly, instead it should be used as a parent class for a database specific query builder, for example *SqliteQueryBuilder* or *PostgresQueryBuilder*.

To implement a sub-class of *BaseQueryBuilder*, you must:

- Implement all methods marked with `@abstractmethod`, such as *build_query()*, *all()* and *fetch()*
- If your DBMS or it's Python API doesn't follow the default query configuration (see the attributes starting with `Q_`), then you should adjust the `Q_` attributes in your class to match your DB / DB API.

  e.g. Set `Q_DEFAULT_PLACEHOLDER = "?"` if your DB API expects `?` for prepared statement placeholders instead of `%s`.
- While not required, you may wish to implement a constructor (*__init__()*), and override *get_cursor()* to adjust it to your database API requirements

abstract all(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[Iterable[dict], Iterable[tuple]]

Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = BaseQueryBuilder('people').all()    # Equivalent to ``SELECT *␣
↪FROM people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop. Ideally, should load rows as you iterate, saving RAM.

abstract build_query() → str

Used internally by *all()* and *fetch()* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

abstract fetch(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]

Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

abstract fetch_next(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]

Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

get_cursor(*cursor_name=None*, *cursor_class=None*, **args*, ***kwargs*)

Create and return a new database cursor object.

It's recommended to override this method if you're inheriting from this class, as this Generic version of `get_cursor` does not make use of `cursor_name` nor `cursor_class`.

Parameters

- **cursor_name** (*str*) – (If DB API supports it) The name for this cursor
- **cursor_class** (*type*) – (If DB API supports it) The cursor class to use

Return GenericCursor cursor A cursor object which should implement at least the basic Python DB API Cursor functionality as specified in *GenericCursor* ((PEP 249)

group_by(**args*)

Add one or more columns to group by clause.

example: group_by('name', 'date') == GROUP BY name, date

Parameters **args** – One or more columns to group by

Returns QueryBuilder object (for chaining)

limit(*limit_num*, *offset=None*)

Add a limit/offset. When using offset you should use an ORDER BY to avoid issues. :param limit_num: Amount of rows to limit to :param offset: Offset by this many rows (optional) :return: QueryBuilder object (for chaining)

order(**args*, *direction='DESC'*)

example: order('mycol', 'othercol') == ORDER BY mycol, othercol DESC

Parameters

- **args** – One or more order columns as individual args
- **direction** – Direction to sort

Returns QueryBuilder object (for chaining)

order_by(**args*, ***kwargs*)

Alias of *order()*

select(**args*)

Add columns to select clause, specify as individual args. NOTE: no escaping!

example:

q.select('mycol', 'othercol', 'somecol as thiscol')

can also chain: q.select('mycol').select('othercol')

Parameters **args** – columns to select as individual arguments

Returns QueryBuilder object (for chaining)

where(*col*, *val*, *compare='='*, *placeholder=None*)
For adding a simple col=value clause with "AND" before it (if at least 1 other clause). val is escaped properly

example: where('x','test').where('y','thing') produces prepared sql "WHERE x = %s AND y = %s"

Parameters

- **col** – the column, function etc. to query
- **val** – the value it should be equal to. most python objects will be converted and escaped properly
- **compare** – instead of '=', compare using this comparator, e.g. '>', '<=' etc.
- **placeholder** – Set the value placeholder, e.g. placeholder='HOST(%s)'

Returns QueryBuilder object (for chaining)

where_or(*col*, *val*, *compare='='*, *placeholder=None*)
For adding simple col=value clause with "OR" before it (if at least 1 other clause). val is escaped properly

example: where('x','test').where_or('y','thing') produces prepared sql "WHERE x = %s OR y = %s"

Parameters

- **col** – the column, function etc. to query
- **val** – the value it should be equal to. most python objects will be converted and escaped properly
- **compare** – instead of '=', compare using this comparator, e.g. '>', '<=' etc.
- **placeholder** – Set the value placeholder, e.g. placeholder='HOST(%s)'

Returns QueryBuilder object (for chaining)

class tests.base.**SqliteQueryBuilder**(*table: str*, *connection: privex.db.types.GenericConnection = None*, ***kwargs*)

all(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[Iterable[dict], Iterable[tuple]]
Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)

Usage:

```
>>> results = BaseQueryBuilder('people').all()    # Equivalent to ``SELECT *␣
↪FROM people;``
>>> for r in results:
>>>     print(r['first_name'], r['last_name'], r['phone'])
```

Return Iterable A cursor which can be iterated using a `for` loop. Ideally, should load rows as you iterate, saving RAM.

build_query() → str
Used internally by *`all()`* and *`fetch()`* - builds and returns a string SQL query using the various class attributes such as `where_clauses` :return str query: The SQL query that will be sent to the database as a string

fetch(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Executes the current query, and fetches the first result as a `dict`.

If there are no results, will return None

Return dict The query result as a dictionary: {column: value, }

Return None If no results are found

fetch_next(*query_mode=<QueryMode.ROW_DICT: 'dict'>*) → Union[dict, tuple, None]
Similar to *fetch()*, but doesn't close the cursor after the query, so can be ran more than once to iterate row-by-row over the results.

Parameters **query_mode** (QueryMode) –

Returns

class tests.base.**QueryMode**
A small enum.Enum used for the query_mode (whether to return rows as tuples or dicts) with Query Builder classes (see *BaseQueryBuilder SqliteQueryBuilder PostgresQueryBuilder*)

class tests.base.**ExampleWrapper**(**args*, ***kwargs*)

class tests.base.**User**(**args*, ***kwargs*)

class tests.base.**ExampleAsyncWrapper**(**args*, ***kwargs*)

2.9.2 tests.test_postgres

Tests related to *PostgresQueryBuilder*, *ExamplePostgresWrapper* and *PostgresWrapper*

Classes

BasePostgresTest([methodName])	Shared base class for Postgres tests.
ExamplePostgresWrapper(*args, **kwargs)	A wrapper around *PostgresWrapper* and TestWrapperMixin for use in Postgres tests.
TestPostgresBuilder([methodName])	
TestPostgresWrapper([methodName])	

2.9.2.1 BasePostgresTest

class tests.test_postgres.**BasePostgresTest**(*methodName='runTest'*)
Shared base class for Postgres tests.

- Sets up *ExamplePostgresWrapper* into the attribute *wrp*
- Before each test (*setUp()*) it deletes and recreates the tables to avoid leftover tables/rows
- After each test (*tearDown()*) it drops all tables to ensure no leftover tables once the tests are done.

__init__(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

conn = None
Holds the module's Postgres connection

setUp() → None
Deletes and recreates all tables to avoid leftover tables/rows

classmethod setUpClass()
Sets up a *ExamplePostgresWrapper* instance under *wrp* for use by tests

tearDown() → None
Deletes all tables after each test finises to avoid leftover tables/rows

wrp: ExamplePostgresWrapper = None
Holds a *ExamplePostgresWrapper* instance for use by test cases

2.9.2.1.1 Methods

Methods

setUp()	Deletes and recreates all tables to avoid leftover tables/rows
setUpClass()	Sets up a *ExamplePostgresWrapper* instance under *wrp* for use by tests
tearDown()	Deletes all tables after each test finises to avoid leftover tables/rows

2.9.2.1.1.1 setUp

BasePostgresTest.**setUp**() → None
Deletes and recreates all tables to avoid leftover tables/rows

2.9.2.1.1.2 setUpClass

classmethod BasePostgresTest.**setUpClass**()
Sets up a *ExamplePostgresWrapper* instance under *wrp* for use by tests

2.9.2.1.1.3 tearDown

BasePostgresTest.**tearDown**() → None
Deletes all tables after each test finises to avoid leftover tables/rows

2.9.2.1.2 Attributes

Attributes

conn	Holds the module's Postgres connection

2.9.2.1.2.1 conn

`BasePostgresTest`.**conn = None**
Holds the module's Postgres connection

2.9.2.2 ExamplePostgresWrapper

class tests.test_postgres.**ExamplePostgresWrapper**(**args*, ***kwargs*)
A wrapper around *PostgresWrapper* and `TestWrapperMixin` for use in Postgres tests.

- Sets the default database to `privex_py_db`
- Creates the table `users`
- Includes two helper query methods *insert_user()* and *find_user()*

__init__(**args*, ***kwargs*)
Initialise the database wrapper class.

Parameters

- **db** (*str*) – Database name
- **db_user** (*str*) – Account username with permission for `db` (defaults to `root`)
- **db_pass** (*str*) – Account password for `db_user` (defaults to `None`)
- **db_host** (*str*) – Database host (defaults to unix socket)

Key str db_schema (Default: `'public'`) Schema used for querying table existence

Key str query_mode Either `'flat'` (query returns tuples) or `'dict'` (query returns dicts). More details in PyDoc block under *query_mode*

Key psycopg2.extensions.cursor cursor_cls If necessary, you may override the Psycopg2 cursor class used by specifying this kwarg. If this isn't specified, *cursor_cls* will default to either `psycopg2.extras.RealDictCursor` if query_mode is `dict`, or `psycopg2.extras.NamedTupleCursor` if query_mode is `flat`.

2.9.2.2.1 Methods

Methods

__init__(*args, **kwargs)	Initialise the database wrapper class.
find_user(id)	
insert_user(first_name, last_name)	

2.9.2.2.1.1 __init__

ExamplePostgresWrapper.**__init__**(**args*, ***kwargs*)

Initialise the database wrapper class.

Parameters

- **db** (*str*) – Database name
- **db_user** (*str*) – Account username with permission for db (defaults to root)
- **db_pass** (*str*) – Account password for db_user (defaults to None)
- **db_host** (*str*) – Database host (defaults to unix socket)

Key str db_schema (Default: 'public') Schema used for querying table existence

Key str query_mode Either 'flat' (query returns tuples) or 'dict' (query returns dicts). More details in PyDoc block under *query_mode*

Key psycopg2.extensions.cursor cursor_cls If necessary, you may override the Psycopg2 cursor class used by specifying this kwarg. If this isn't specified, *cursor_cls* will default to either psycopg2.extras.RealDictCursor if query_mode is dict, or psycopg2.extras.NamedTupleCursor if query_mode is flat.

2.9.2.2.1.2 find_user

ExamplePostgresWrapper.**find_user**(*id: int*)

2.9.2.2.1.3 insert_user

ExamplePostgresWrapper.**insert_user**(*first_name*, *last_name*)

2.9.2.2.2 Attributes

Attributes

DEFAULT_DB
SCHEMAS

2.9.2.2.2.1 DEFAULT_DB

ExamplePostgresWrapper.**DEFAULT_DB: str = 'privex_py_db'**

2.9.2.2.2.2 SCHEMAS

ExamplePostgresWrapper.**SCHEMAS: List[Tuple[str, str]] = [('users', 'CREATE TABLE users (id**

2.9.2.3 TestPostgresBuilder

class tests.test_postgres.**TestPostgresBuilder**(*methodName='runTest'*)

__init__(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_all_call()
Insert two users, then verify they're returned from an `.all()` call using *PostgresQueryBuilder*

test_generator_builder()
Test obtaining PostgresBuilder results by calling `next()` on the builder object (like a generator)

test_group_call()
Insert 5 users with 3 "John"s, then run a select+where+group_by query and confirm COUNT returns 3 johns

test_index_builder()
Test obtaining PostgresBuilder results by accessing an index of the builder object

test_iterate_builder()
Test obtaining PostgresBuilder results by iterating over the builder object itself with a for loop

test_query_all()
Build a select all query using *PostgresQueryBuilder* and confirm the built query looks correct

test_query_select_col_where()
Build a query selecting specific columns plus a 'where AND' clause using *PostgresQueryBuilder* and confirm the built query looks correct

test_query_select_col_where_group()
Build a complex select+where+group_by query using *PostgresQueryBuilder* and confirm the built query looks correct

test_query_select_col_where_order()
Build a query selecting specific columns, a 'where AND' clause, and an 'ORDER BY' clause using *PostgresQueryBuilder* and confirm the built query looks correct

test_query_where_first_name_last_name()
Build a 'where AND' query using *PostgresQueryBuilder* and confirm the built query looks correct

test_where_call()
Insert three users, then retrieve Dave using `.where()` + `.fetch()` call on *PostgresQueryBuilder*

2.9.2.3.1 Methods

Methods

test_all_call()	Insert two users, then verify they're returned from an `.all()` call using *PostgresQueryBuilder*
test_generator_builder()	Test obtaining PostgresBuilder results by calling `next()` on the builder object (like a generator)
test_group_call()	Insert 5 users with 3 "John"s, then run a select+where+group_by query and confirm COUNT returns 3 johns
test_index_builder()	Test obtaining PostgresBuilder results by accessing an index of the builder object
test_iterate_builder()	Test obtaining PostgresBuilder results by iterating over the builder object itself with a for loop
test_query_all()	Build a select all query using *PostgresQueryBuilder* and confirm the built query looks correct
test_query_select_col_where()	Build a query selecting specific columns plus a 'where AND' clause using *PostgresQueryBuilder* and confirm the built query looks correct
test_query_select_col_where_group()	Build a complex select+where+group_by query using *PostgresQueryBuilder* and confirm the built query looks correct
test_query_select_col_where_order()	Build a query selecting specific columns, a 'where AND' clause, and an 'ORDER BY' clause using *PostgresQueryBuilder* and confirm the built query looks correct
test_query_where_first_name_last_name()	Build a 'where AND' query using *PostgresQueryBuilder* and confirm the built query looks correct
test_where_call()	Insert three users, then retrieve Dave using `.where()` + `.fetch()` call on *PostgresQueryBuilder*

2.9.2.3.1.1 test_all_call

`TestPostgresBuilder.`**`test_all_call`**`()`

Insert two users, then verify they're returned from an `.all()` call using *PostgresQueryBuilder*

2.9.2.3.1.2 test_generator_builder

TestPostgresBuilder.**test_generator_builder**()
Test obtaining PostgresBuilder results by calling `next()` on the builder object (like a generator)

2.9.2.3.1.3 test_group_call

TestPostgresBuilder.**test_group_call**()
Insert 5 users with 3 "John"s, then run a select+where+group_by query and confirm COUNT returns 3 johns

2.9.2.3.1.4 test_index_builder

TestPostgresBuilder.**test_index_builder**()
Test obtaining PostgresBuilder results by accessing an index of the builder object

2.9.2.3.1.5 test_iterate_builder

TestPostgresBuilder.**test_iterate_builder**()
Test obtaining PostgresBuilder results by iterating over the builder object itself with a for loop

2.9.2.3.1.6 test_query_all

TestPostgresBuilder.**test_query_all**()
Build a select all query using *PostgresQueryBuilder* and confirm the built query looks correct

2.9.2.3.1.7 test_query_select_col_where

TestPostgresBuilder.**test_query_select_col_where**()
Build a query selecting specific columns plus a 'where AND' clause using *PostgresQueryBuilder* and confirm the built query looks correct

2.9.2.3.1.8 test_query_select_col_where_group

TestPostgresBuilder.**test_query_select_col_where_group**()
Build a complex select+where+group_by query using *PostgresQueryBuilder* and confirm the built query looks correct

2.9.2.3.1.9 test_query_select_col_where_order

TestPostgresBuilder.**test_query_select_col_where_order**()
Build a query selecting specific columns, a 'where AND' clause, and an 'ORDER BY' clause using *PostgresQueryBuilder* and confirm the built query looks correct

2.9.2.3.1.10 test_query_where_first_name_last_name

TestPostgresBuilder.**test_query_where_first_name_last_name**()
Build a 'where AND' query using *PostgresQueryBuilder* and confirm the built query looks correct

2.9.2.3.1.11 test_where_call

TestPostgresBuilder.**test_where_call**()
Insert three users, then retrieve Dave using `.where()` + `.fetch()` call on *PostgresQueryBuilder*

2.9.2.3.2 Attributes

Attributes

pytestmark

2.9.2.3.2.1 pytestmark

TestPostgresBuilder.**pytestmark = [Mark(name='skipif', args=(False,), kwargs={'reason': "L**

2.9.2.4 TestPostgresWrapper

class tests.test_postgres.**TestPostgresWrapper**(*methodName='runTest'*)

__init__(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

2.9.2.4.1 Methods

Methods

test_action_update()
test_find_user_dict_mode()
test_find_user_nonexistent()
test_get_users_dict()
test_get_users_tuple()
test_insert_find_user()
test_tables_created()
test_tables_drop()

2.9.2.4.1.1 test_action_update

TestPostgresWrapper.**test_action_update**()

2.9.2.4.1.2 test_find_user_dict_mode

TestPostgresWrapper.**test_find_user_dict_mode**()

2.9.2.4.1.3 test_find_user_nonexistent

TestPostgresWrapper.**test_find_user_nonexistent**()

2.9.2.4.1.4 test_get_users_dict

TestPostgresWrapper.**test_get_users_dict**()

2.9.2.4.1.5 test_get_users_tuple

TestPostgresWrapper.**test_get_users_tuple**()

2.9.2.4.1.6 test_insert_find_user

TestPostgresWrapper.**test_insert_find_user**()

2.9.2.4.1.7 test_tables_created

TestPostgresWrapper.**test_tables_created**()

2.9.2.4.1.8 test_tables_drop

TestPostgresWrapper.**test_tables_drop**()

2.9.2.4.2 Attributes

Attributes

pytestmark

2.9.2.4.2.1 pytestmark

`TestPostgresWrapper.pytestmark = [Mark(name='skipif', args=(False,), kwargs={'reason': "L`

class tests.test_postgres.**BasePostgresTest**(*methodName='runTest'*)

Shared base class for Postgres tests.

- Sets up *ExamplePostgresWrapper* into the attribute *wrp*
- Before each test (*setUp()*) it deletes and recreates the tables to avoid leftover tables/rows
- After each test (*tearDown()*) it drops all tables to ensure no leftover tables once the tests are done.

conn = None

Holds the module's Postgres connection

setUp() → None

Deletes and recreates all tables to avoid leftover tables/rows

classmethod setUpClass()

Sets up a *ExamplePostgresWrapper* instance under *wrp* for use by tests

tearDown() → None

Deletes all tables after each test finises to avoid leftover tables/rows

wrp: ExamplePostgresWrapper = None

Holds a *ExamplePostgresWrapper* instance for use by test cases

class tests.test_postgres.**ExamplePostgresWrapper**(**args*, ***kwargs*)

A wrapper around *PostgresWrapper* and `TestWrapperMixin` for use in Postgres tests.

- Sets the default database to `privex_py_db`
- Creates the table `users`
- Includes two helper query methods *insert_user()* and *find_user()*

class tests.test_postgres.**TestPostgresBuilder**(*methodName='runTest'*)

test_all_call()

Insert two users, then verify they're returned from an `.all()` call using *PostgresQueryBuilder*

test_generator_builder()

Test obtaining PostgresBuilder results by calling `next()` on the builder object (like a generator)

test_group_call()

Insert 5 users with 3 "John"s, then run a select+where+group_by query and confirm COUNT returns 3 johns

test_index_builder()

Test obtaining PostgresBuilder results by accessing an index of the builder object

test_iterate_builder()

Test obtaining PostgresBuilder results by iterating over the builder object itself with a for loop

test_query_all()

Build a select all query using *PostgresQueryBuilder* and confirm the built query looks correct

test_query_select_col_where()

Build a query selecting specific columns plus a 'where AND' clause using *PostgresQueryBuilder* and confirm the built query looks correct

test_query_select_col_where_group()
Build a complex select+where+group_by query using *PostgresQueryBuilder* and confirm the built query looks correct

test_query_select_col_where_order()
Build a query selecting specific columns, a 'where AND' clause, and an 'ORDER BY' clause using *PostgresQueryBuilder* and confirm the built query looks correct

test_query_where_first_name_last_name()
Build a 'where AND' query using *PostgresQueryBuilder* and confirm the built query looks correct

test_where_call()
Insert three users, then retrieve Dave using `.where()` + `.fetch()` call on *PostgresQueryBuilder*

class tests.test_postgres.**TestPostgresWrapper**(*methodName='runTest'*)

2.9.3 tests.test_sqlite_builder

Tests related to *SqliteQueryBuilder* and *ExampleWrapper*

Classes

TestSQLiteBuilder([methodName])	

2.9.3.1 TestSQLiteBuilder

class tests.test_sqlite_builder.**TestSQLiteBuilder**(*methodName='runTest'*)

__init__(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

test_generator_builder()
Test obtaining SqliteQueryBuilder results by calling `next()` on the builder object (like a generator)

test_index_builder()
Test obtaining SqliteQueryBuilder results by accessing an index of the builder object

test_iterate_builder()
Test obtaining SqliteQueryBuilder results by iterating over the builder object itself with a for loop

2.9.3.1.1 Methods

Methods

test_all_call()	
test_generator_builder()	Test obtaining SqliteQueryBuilder results by calling `next()` on the builder object (like a generator)
test_group_call()	

Continued on next page

Table 47 – continued from previous page

test_index_builder()	Test obtaining SqliteQueryBuilder results by accessing an index of the builder object
test_iterate_builder()	Test obtaining SqliteQueryBuilder results by iterating over the builder object itself with a for loop
test_query_all()	
test_query_select_col_where()	
test_query_select_col_where_group()	
test_query_select_col_where_order()	
test_query_where_first_name_last_name()	
test_where_call()	

2.9.3.1.1.1 test_all_call

TestSQLiteBuilder.**test_all_call**()

2.9.3.1.1.2 test_generator_builder

TestSQLiteBuilder.**test_generator_builder**()
Test obtaining SqliteQueryBuilder results by calling `next()` on the builder object (like a generator)

2.9.3.1.1.3 test_group_call

TestSQLiteBuilder.**test_group_call**()

2.9.3.1.1.4 test_index_builder

TestSQLiteBuilder.**test_index_builder**()
Test obtaining SqliteQueryBuilder results by accessing an index of the builder object

2.9.3.1.1.5 test_iterate_builder

TestSQLiteBuilder.**test_iterate_builder**()
Test obtaining SqliteQueryBuilder results by iterating over the builder object itself with a for loop

2.9.3.1.1.6 test_query_all

TestSQLiteBuilder.**test_query_all**()

2.9.3.1.1.7 test_query_select_col_where

TestSQLiteBuilder.**test_query_select_col_where**()

2.9.3.1.1.8 test_query_select_col_where_group

TestSQLiteBuilder.**test_query_select_col_where_group**()

2.9.3.1.1.9 test_query_select_col_where_order

TestSQLiteBuilder.**test_query_select_col_where_order**()

2.9.3.1.1.10 test_query_where_first_name_last_name

TestSQLiteBuilder.**test_query_where_first_name_last_name**()

2.9.3.1.1.11 test_where_call

TestSQLiteBuilder.**test_where_call**()

2.9.3.1.2 Attributes

Attributes

---

class tests.test_sqlite_builder.**TestSQLiteBuilder**(*methodName='runTest'*)

test_generator_builder()
Test obtaining SqliteQueryBuilder results by calling `next()` on the builder object (like a generator)

test_index_builder()
Test obtaining SqliteQueryBuilder results by accessing an index of the builder object

test_iterate_builder()
Test obtaining SqliteQueryBuilder results by iterating over the builder object itself with a for loop

2.9.4 tests.test_sqlite_wrapper

Tests related to *SqliteWrapper* / *ExampleWrapper*

Classes

TestSQLiteWrapper([methodName])

2.9.4.1 TestSQLiteWrapper

class tests.test_sqlite_wrapper.**TestSQLiteWrapper**(*methodName='runTest'*)

__init__(*methodName='runTest'*)
Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

2.9.4.1.1 Methods

Methods

test_action_update()
test_find_user_dict_mode()
test_find_user_nonexistent()
test_get_users_dict()
test_get_users_tuple()
test_insert_find_user()
test_tables_created()
test_tables_drop()

2.9.4.1.1.1 test_action_update

TestSQLiteWrapper.**test_action_update**()

2.9.4.1.1.2 test_find_user_dict_mode

TestSQLiteWrapper.**test_find_user_dict_mode**()

2.9.4.1.1.3 test_find_user_nonexistent

TestSQLiteWrapper.**test_find_user_nonexistent**()

2.9.4.1.1.4 test_get_users_dict

TestSQLiteWrapper.**test_get_users_dict**()

2.9.4.1.1.5 test_get_users_tuple

TestSQLiteWrapper.**test_get_users_tuple**()

2.9.4.1.1.6 test_insert_find_user

TestSQLiteWrapper.**test_insert_find_user**()

2.9.4.1.1.7 test_tables_created

TestSQLiteWrapper.**test_tables_created**()

2.9.4.1.1.8 test_tables_drop

TestSQLiteWrapper.**test_tables_drop**()

2.9.4.1.2 Attributes

Attributes

---

class tests.test_sqlite_wrapper.**TestSQLiteWrapper**(*methodName='runTest'*)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

t

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

O

P

Q

R

S

T

U

W