
Privex Python Database Wrappers Documentation

Privex Inc., Chris (Someguy123)

Dec 19, 2019

MAIN:

1	Quick install	3
2	All Documentation	5
2.1	Installing Privex Python DB Wrappers	5
2.1.1	Download and install from PyPi using pipenv / pip (recommended)	5
2.1.2	(Alternative) Manual install from Git	5
2.2	Examples / Basic Usage	6
2.2.1	Using the SQLite3 Manager + Query Builder	6
2.2.1.1	Basic / direct usage of SqliteWrapper	6
2.2.1.2	Using the query builder (SqliteQueryBuilder)	6
2.2.1.3	Sub-classing SqliteWrapper for your app	7
2.2.1.4	Using the query builder from your sub-class	8
2.3	privex.db.base	9
2.3.1	CursorManager	10
2.3.1.1	Methods	10
2.3.1.1.1	__init__	11
2.3.1.1.2	close	11
2.3.1.1.3	execute	11
2.3.1.1.4	fetchall	11
2.3.1.1.5	fetchmany	11
2.3.1.1.6	fetchone	11
2.3.1.2	Attributes	11
2.3.1.2.1	description	12
2.3.1.2.2	lastrowid	12
2.3.1.2.3	rowcount	12
2.3.2	DBExecution	12
2.3.2.1	Methods	12
2.3.2.2	Attributes	12
2.3.3	GenericDBWrapper	12
2.3.3.1	Methods	19
2.3.3.1.1	__init__	20
2.3.3.1.2	action	20
2.3.3.1.3	close_cursor	21
2.3.3.1.4	create_schema	21
2.3.3.1.5	create_schemas	21
2.3.3.1.6	drop_schemas	21
2.3.3.1.7	drop_table	22
2.3.3.1.8	drop_tables	22
2.3.3.1.9	fetch	22
2.3.3.1.10	fetchall	23

	2.3.3.1.11	fetchone	23
	2.3.3.1.12	get_cursor	23
	2.3.3.1.13	list_tables	23
	2.3.3.1.14	make_connection	24
	2.3.3.1.15	query	24
	2.3.3.1.16	recreate_schemas	25
	2.3.3.1.17	table_exists	26
	2.3.3.2	Attributes	26
	2.3.3.2.1	AUTO_ZIP_COLS	26
	2.3.3.2.2	DEFAULT_ENABLE_EXECUTION_LOG	27
	2.3.3.2.3	DEFAULT_QUERY_MODE	27
	2.3.3.2.4	DEFAULT_TABLE_LIST_QUERY	27
	2.3.3.2.5	DEFAULT_TABLE_QUERY	27
	2.3.3.2.6	SCHEMAS	27
	2.3.3.2.7	conn	27
	2.3.3.2.8	cursor	28
	2.3.3.2.9	tables_created	28
2.4	privex.db.postgres		35
	2.4.1	PostgresWrapper	35
	2.4.1.1	Methods	39
	2.4.1.1.1	__init__	39
	2.4.1.1.2	builder	40
	2.4.1.1.3	drop_table	40
	2.4.1.1.4	last_insert_id	40
	2.4.1.1.5	list_tables	40
	2.4.1.1.6	query	41
	2.4.1.1.7	table_exists	42
	2.4.1.2	Attributes	43
	2.4.1.2.1	AUTO_ZIP_COLS	43
	2.4.1.2.2	DEFAULT_DB	43
	2.4.1.2.3	DEFAULT_QUERY_MODE	43
	2.4.1.2.4	DEFAULT_TABLE_LIST_QUERY	43
	2.4.1.2.5	DEFAULT_TABLE_QUERY	43
	2.4.1.2.6	conn	43
	2.4.1.2.7	cursor_cls	44
	2.4.1.2.8	cursor_map	44
2.5	privex.db.sqlite		47
	2.5.1	SqliteWrapper	48
	2.5.1.1	Methods	50
	2.5.1.1.1	__init__	50
	2.5.1.1.2	builder	50
	2.5.1.2	Attributes	50
	2.5.1.2.1	DEFAULT_DB	51
	2.5.1.2.2	DEFAULT_DB_FOLDER	51
	2.5.1.2.3	DEFAULT_DB_NAME	51
	2.5.1.2.4	DEFAULT_TABLE_LIST_QUERY	51
	2.5.1.2.5	DEFAULT_TABLE_QUERY	51
	2.5.1.2.6	conn	51
2.6	privex.db.types		56
	2.6.1	GenericConnection	57
	2.6.1.1	Methods	57
	2.6.1.1.1	__init__	57
	2.6.1.1.2	close	57
	2.6.1.1.3	commit	58

	2.6.1.1.4	cursor	58
	2.6.1.1.5	rollback	58
2.6.2	GenericCursor		58
2.6.2.1	Methods		58
	2.6.2.1.1	__init__	58
	2.6.2.1.2	close	59
	2.6.2.1.3	execute	59
	2.6.2.1.4	executemany	59
	2.6.2.1.5	fetchall	59
	2.6.2.1.6	fetchmany	59
	2.6.2.1.7	fetchone	59
2.7	privex.db.query		59
2.7.1	privex.db.query.base		60
2.7.1.1	BaseQueryBuilder		60
2.7.1.1.1	Methods		62
	2.7.1.1.1.1	__init__	63
	2.7.1.1.1.2	all	63
	2.7.1.1.1.3	build_query	64
	2.7.1.1.1.4	close_cursor	64
	2.7.1.1.1.5	execute	64
	2.7.1.1.1.6	fetch	64
	2.7.1.1.1.7	fetch_next	64
	2.7.1.1.1.8	get_cursor	64
	2.7.1.1.1.9	group_by	65
	2.7.1.1.1.10	limit	65
	2.7.1.1.1.11	order	65
	2.7.1.1.1.12	order_by	65
	2.7.1.1.1.13	select	65
	2.7.1.1.1.14	where	66
	2.7.1.1.1.15	where_or	66
2.7.1.1.2	Attributes		66
	2.7.1.1.2.1	Q_DEFAULT_PLACEHOLDER	67
	2.7.1.1.2.2	Q_GROUP_BY_CLAUSE	67
	2.7.1.1.2.3	Q_LIMIT_CLAUSE	67
	2.7.1.1.2.4	Q_OFFSET_CLAUSE	67
	2.7.1.1.2.5	Q_ORDER_CLAUSE	67
	2.7.1.1.2.6	Q_POST_QUERY	67
	2.7.1.1.2.7	Q_PRE_QUERY	67
	2.7.1.1.2.8	Q_SELECT_CLAUSE	67
	2.7.1.1.2.9	Q_WHERE_CLAUSE	67
	2.7.1.1.2.10	connection	68
	2.7.1.1.2.11	cursor	68
2.7.1.2	QueryMode		68
2.7.1.2.1	Attributes		68
	2.7.1.2.1.1	DEFAULT	68
	2.7.1.2.1.2	ROW_DICT	68
	2.7.1.2.1.3	ROW_TUPLE	68
2.7.2	privex.db.query.postgres		71
2.7.2.1	PostgresQueryBuilder		71
2.7.2.1.1	Methods		73
	2.7.2.1.1.1	__init__	74
	2.7.2.1.1.2	all	74
	2.7.2.1.1.3	build_query	74
	2.7.2.1.1.4	fetch	74

	2.7.2.1.1.5	fetch_next	75
	2.7.2.1.1.6	get_cursor	75
	2.7.2.1.1.7	query_mode_cursor	75
	2.7.2.1.1.8	select_date	75
	2.7.2.1.2	Attributes	76
	2.7.2.1.2.1	Q_DEFAULT_PLACEHOLDER	76
	2.7.2.1.2.2	Q_PRE_QUERY	76
	2.7.2.1.2.3	conn	76
	2.7.2.1.2.4	cursor	76
2.7.3	privex.db.query.sqlite		78
2.7.3.1	SQLiteQueryBuilder		79
2.7.3.1.1	Methods		79
	2.7.3.1.1.1	all	80
	2.7.3.1.1.2	build_query	80
	2.7.3.1.1.3	fetch	80
	2.7.3.1.1.4	fetch_next	80
2.7.3.1.2	Attributes		81
	2.7.3.1.2.1	Q_DEFAULT_PLACEHOLDER	81
	2.7.3.1.2.2	Q_PRE_QUERY	81
	2.7.3.1.2.3	conn	81
	2.7.3.1.2.4	connection	81
2.8	How to use the unit tests		82
2.8.1	Testing pre-requisites		82
2.8.2	Running via PyTest		82
2.8.3	Running individual test modules		83
2.9	Unit Test List / Overview		84
2.9.1	tests.base		85
2.9.1.1	ExampleWrapper		85
2.9.1.1.1	Methods		86
	2.9.1.1.1.1	__init__	86
2.9.1.1.2	Attributes		86
	2.9.1.1.2.1	DEFAULT_DB	86
	2.9.1.1.2.2	SCHEMAS	87
2.9.1.2	PrivexDBTestBase		87
2.9.1.2.1	Methods		87
	2.9.1.2.1.1	setUp	87
	2.9.1.2.1.2	tearDown	87
2.9.1.2.2	Attributes		87
2.9.1.3	User		88
	2.9.1.3.1	Methods	88
	2.9.1.3.2	Attributes	88
2.9.2	tests.test_postgres		93
2.9.2.1	BasePostgresTest		93
2.9.2.1.1	Methods		94
	2.9.2.1.1.1	setUp	94
	2.9.2.1.1.2	setUpClass	94
	2.9.2.1.1.3	tearDown	94
2.9.2.1.2	Attributes		94
	2.9.2.1.2.1	conn	95
2.9.2.2	ExamplePostgresWrapper		95
2.9.2.2.1	Methods		95
	2.9.2.2.1.1	__init__	96
	2.9.2.2.1.2	find_user	96
	2.9.2.2.1.3	insert_user	96

2.9.2.2.2	Attributes	96
2.9.2.2.2.1	DEFAULT_DB	96
2.9.2.2.2.2	SCHEMAS	97
2.9.2.3	TestPostgresBuilder	97
2.9.2.3.1	Methods	98
2.9.2.3.1.1	test_all_call	98
2.9.2.3.1.2	test_generator_builder	99
2.9.2.3.1.3	test_group_call	99
2.9.2.3.1.4	test_index_builder	99
2.9.2.3.1.5	test_iterate_builder	99
2.9.2.3.1.6	test_query_all	99
2.9.2.3.1.7	test_query_select_col_where	99
2.9.2.3.1.8	test_query_select_col_where_group	99
2.9.2.3.1.9	test_query_select_col_where_order	99
2.9.2.3.1.10	test_query_where_first_name_last_name	100
2.9.2.3.1.11	test_where_call	100
2.9.2.3.2	Attributes	100
2.9.2.3.2.1	pytestmark	100
2.9.2.4	TestPostgresWrapper	100
2.9.2.4.1	Methods	100
2.9.2.4.1.1	test_action_update	101
2.9.2.4.1.2	test_find_user_dict_mode	101
2.9.2.4.1.3	test_find_user_nonexistent	101
2.9.2.4.1.4	test_get_users_dict	101
2.9.2.4.1.5	test_get_users_tuple	101
2.9.2.4.1.6	test_insert_find_user	101
2.9.2.4.1.7	test_tables_created	101
2.9.2.4.1.8	test_tables_drop	101
2.9.2.4.2	Attributes	101
2.9.2.4.2.1	pytestmark	102
2.9.3	tests.test_sqlite_builder	103
2.9.3.1	TestSQLiteBuilder	103
2.9.3.1.1	Methods	103
2.9.3.1.1.1	test_all_call	104
2.9.3.1.1.2	test_generator_builder	104
2.9.3.1.1.3	test_group_call	104
2.9.3.1.1.4	test_index_builder	104
2.9.3.1.1.5	test_iterate_builder	104
2.9.3.1.1.6	test_query_all	104
2.9.3.1.1.7	test_query_select_col_where	105
2.9.3.1.1.8	test_query_select_col_where_group	105
2.9.3.1.1.9	test_query_select_col_where_order	105
2.9.3.1.1.10	test_query_where_first_name_last_name	105
2.9.3.1.1.11	test_where_call	105
2.9.3.1.2	Attributes	105
2.9.4	tests.test_sqlite_wrapper	106
2.9.4.1	TestSQLiteWrapper	106
2.9.4.1.1	Methods	106
2.9.4.1.1.1	test_action_update	106
2.9.4.1.1.2	test_find_user_dict_mode	107
2.9.4.1.1.3	test_find_user_nonexistent	107
2.9.4.1.1.4	test_get_users_dict	107
2.9.4.1.1.5	test_get_users_tuple	107
2.9.4.1.1.6	test_insert_find_user	107

2.9.4.1.1.7	test_tables_created	107
2.9.4.1.1.8	test_tables_drop	107
2.9.4.1.2	Attributes	107
3	Indices and tables	109
	Python Module Index	111
	Index	113

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

- *Privex Python Database Wrappers (privex-db) documentation*
 - *Quick install*
- *All Documentation*
- *Indices and tables*

QUICK INSTALL

Installing with [Pipenv](#) (recommended)

```
pipenv install privex-db
```

Installing with standard `pip3`

```
pip3 install privex-db
```


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}
```

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

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

<code>CursorManager(cursor[, close_callback])</code>	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 <code>__enter__()</code> and <code>__exit__()</code>
<code>DBExecution(*args, **kwargs)</code>	
<code>GenericDBWrapper([db, connector_func])</code>	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 = None
```

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

```
    is_context_manager = None
```

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

2.3.1.1 Methods

Methods

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

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 be 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 = 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 = '?'

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

DEFAULT_QUERY_MODE = '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 = []

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, **kwargs*) → 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, **kwargs)*
- **kwargs** – Extra arguments will be passed through to *cursor.execute(sql, *params, **kwargs)*

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/kwarg 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', **kwargs*) → 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`).
- **kwargs** – 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, **kwargs)`. 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 = 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 = {}

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

<code>__init__([db, connector_func])</code>	Initialise the database wrapper class.
<code>action(sql, *params, **kwargs)</code>	Use <code>action()</code> as a simple alias method for running “action” queries which don’t return results, only affected row counts.
<code>close_cursor()</code>	
<code>create_schema(table[, schema])</code>	Create the individual table <code>table</code> , either uses the create statement <code>schema</code> , or if <code>schema</code> is empty, then checks for a pre-existing CREATE statement for <code>table</code> in <code>SCHEMAS</code> .
<code>create_schemas(*tables)</code>	Create all tables listed in <code>SCHEMAS</code> if they don’t already exist.
<code>drop_schemas(*tables)</code>	Drop all tables listed in <code>SCHEMAS</code> if they exist.
<code>drop_table(table)</code>	Drop the table <code>table</code> if it exists.
<code>drop_tables(*tables)</code>	Drop one or more tables as positional arguments.
<code>fetch(sql, *params[, fetch])</code>	Similar to <code>query()</code> but only returns the fetch results, not the execution object nor cursor.
<code>fetchall(sql, *params, **kwargs)</code>	Alias for <code>fetch()</code> with <code>fetch='all'</code>
<code>fetchone(sql, *params, **kwargs)</code>	Alias for <code>fetch()</code> with <code>fetch='one'</code>
<code>get_cursor([cursor_name, cursor_class])</code>	Create and return a new database cursor object, by default the cursor will be wrapped with <code>CursorManager</code> to ensure context management (with statements) works regardless of whether the database API supports context managing cursors (e.g.
<code>list_tables()</code>	Get a list of tables present in the current database.
<code>make_connection(*args, **kwargs)</code>	Creates a database connection using <code>connector_func</code> , passing all arguments/kwargs directly to the connector function.
<code>query(sql, *params[, fetch])</code>	Create an instance of your database wrapper:
<code>recreate_schemas(*tables)</code>	Drop all tables then re-create them.
<code>table_exists(table)</code>	Returns True if the table <code>table</code> 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 be 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, **kwargs) → Union[List[dict], List[tuple], None]`
 Alias for `fetch()` with `fetch='all'`

2.3.3.1.11 fetchone

`GenericDBWrapper.fetchone (sql: str, *params, **kwargs) → 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`).
- **kwargs** – 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, **kwargs)`. 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.

2.3.3.16 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

<i>AUTO_ZIP_COLS</i>	If your database API doesn't support returning rows as dicts or a dict-like structure (e.g.
<i>DEFAULT_ENABLE_EXECUTION_LOG</i>	
<i>DEFAULT_QUERY_MODE</i>	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.
<i>DEFAULT_TABLE_LIST_QUERY</i>	
<i>DEFAULT_TABLE_QUERY</i>	
<i>SCHEMAS</i>	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.
<i>conn</i>	Get or create a database connection
<i>cursor</i>	
<i>tables_created</i>	This is a static class attribute which tracks which tables have already been created, avoiding <i>create_schemas()</i> having to make as many queries every time the class is constructed.

2.3.3.2.1 AUTO_ZIP_COLS

GenericDBWrapper.**AUTO_ZIP_COLS = 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 = True
```

2.3.3.2.3 DEFAULT_QUERY_MODE

```
GenericDBWrapper.DEFAULT_QUERY_MODE = '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 = []
```

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 = {}`

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 = None

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

is_context_manager = 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 = None

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

is_context_manager = 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 = 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 = '?'

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

DEFAULT_QUERY_MODE = '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 = []

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'*, ***kwargs*) → 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, **kwargs)`
- **kwargs** – Extra arguments will be passed through to `cursor.execute(sql, *params, **kwargs)`

Returns

fetchall (*sql: str*, **params*, ***kwargs*) → Union[List[dict], List[tuple], None]

Alias for `fetch()` with `fetch='all'`

fetchone (*sql: str*, **params*, ***kwargs*) → 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), ('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).
- **kwargs** – 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, **kwargs)`. 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 = 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 = {}
```

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

<code>PostgresWrapper</code> (<code>[db, db_user, db_host, db_pass]</code>)	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 = 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', **kwargs*) → Tuple[Optional[iter], Any, psy-copg2.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), ('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, **kwargs)`. 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

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

<i>AUTO_ZIP_COLS</i>	
<i>DEFAULT_DB</i>	
<i>DEFAULT_QUERY_MODE</i>	
<i>DEFAULT_TABLE_LIST_QUERY</i>	
<i>DEFAULT_TABLE_QUERY</i>	
<i>conn</i>	Get or create a Postgres connection
<i>cursor_cls</i>	
<i>cursor_map</i>	

2.4.1.2.1 AUTO_ZIP_COLS

```
PostgresWrapper.AUTO_ZIP_COLS = False
```

2.4.1.2.2 DEFAULT_DB

```
PostgresWrapper.DEFAULT_DB = 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': <class 'psycogp2.extras.RealDictCursor'>, 'flat': <class 'psycogp2.extras.FlatDictCursor'>}`

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 = 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: psycogp2.extensions.cursor = None, **fields*) →

`Union[privex.helpers.collections.DictObject, psycogp2.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), ('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, **kwargs)`. 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

<code>SqliteWrapper([db, isolation_level])</code>	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 = '/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 = '/home/docs/.privex_sqlite'

If an absolute path isn't given, store the sqlite3 database file in this folder

DEFAULT_DB_NAME = '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 = 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

<code>DEFAULT_TABLE_LIST_QUERY</code>	
<code>DEFAULT_TABLE_QUERY</code>	
<code>conn</code>	Get or create an SQLite3 connection using DB file <code>db</code> and return it

2.5.1.2.1 DEFAULT_DB

`SQLiteWrapper.DEFAULT_DB = '/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 = '/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 = '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 = 'table'"`

2.5.1.2.5 DEFAULT_TABLE_QUERY

`SQLiteWrapper.DEFAULT_TABLE_QUERY = "SELECT count(name) as table_count FROM sqlite_master WHERE type = 'table'"`

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 = '/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 = '/home/docs/.privex_sqlite'

If an absolute path isn't given, store the sqlite3 database file in this folder

DEFAULT_DB_NAME = '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 = 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 = '/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 = '/home/docs/.privex_sqlite'

If an absolute path isn't given, store the sqlite3 database file in this folder

DEFAULT_DB_NAME = '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 = 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

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

<code>__init__(*args, **kwargs)</code>
<code>close(*args, **kwargs)</code>
<code>commit(*args, **kwargs)</code>
<code>cursor(*args, **kwargs)</code>
<code>rollback(*args, **kwargs)</code>

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

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

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

`privex.db.query.base`

`privex.db.query.postgres`

`privex.db.query.sqlite`

2.7.1 privex.db.query.base

Classes

<code>BaseQueryBuilder(table[, connection])</code>	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.
<code>QueryMode</code>	A small <code>enum.Enum</code> used for the <code>query_mode</code> (whether to return rows as tuples or dicts) with Query Builder classes (see <code>BaseQueryBuilder</code> <code>SqliteQueryBuilder</code> <code>PostgresQueryBuilder</code>)

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

<code>__init__(table[, connection])</code>	Initialize self.
<code>all([query_mode])</code>	Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)
<code>build_query()</code>	Used internally by <code>all()</code> and <code>fetch()</code> - builds and returns a string SQL query using the various class attributes such as <code>where_clauses</code> :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

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

<i>PostgresQueryBuilder</i> (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*) → *psy-copg2.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

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

Continued on next page

Table 24 – continued from previous page

<code>query_mode_cursor(query_mode[, ...])</code>	Return a cursor object with the cursor class based on the <code>query_mode</code> , using the <code>query_mode</code> to cursor class map in <code>_cursor_map</code>
<code>select_date(*args)</code>	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

<code>Q_DEFAULT_PLACEHOLDER</code>
<code>Q_PRE_QUERY</code>
<code>conn</code>
<code>cursor</code>

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*) → psy-
copg2.extensions.cursor

Create and return a new Postgres cursor object

query_mode_cursor (*query_mode: privex.db.query.base.QueryMode, replace_cursor=True, cur-
sor_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

<i>all</i> ([<i>query_mode</i>])	Executes the current query, and returns an iterable cursor (results are loaded as you iterate the cursor)
<i>build_query</i> ()	Used internally by <i>all()</i> and <i>fetch()</i> - builds and returns a string SQL query using the various class attributes such as <i>where_clauses</i> :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

<code>fetch([query_mode])</code>	Executes the current query, and fetches the first result as a dict.
<code>fetch_next([query_mode])</code>	Similar to <code>fetch()</code> , 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 = 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

<i>base</i>	This file contains code shared between tests, such as <i>PrivexDBTestBase</i> which is the base class shared by all unit tests.
<i>test_postgres</i>	Tests related to <i>PostgresQueryBuilder</i> , <i>ExamplePostgresWrapper</i> and <i>PostgresWrapper</i>

Continued on next page

Table 29 – continued from previous page

<code>test_sqlite_builder</code>	Tests related to <code>SqliteQueryBuilder</code> and <code>ExampleWrapper</code>
<code>test_sqlite_wrapper</code>	Tests related to <code>SqliteWrapper</code> / <code>ExampleWrapper</code>

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

<code>ExampleWrapper(*args, **kwargs)</code>	
<code>PrivexDBTestBase([methodName])</code>	Base class for all privex-db test classes.
<code>User(*args, **kwargs)</code>	

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 (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'

2.9.1.1.2 Attributes

Attributes

```
DEFAULT_DB
```

```
SCHEMAS
```

2.9.1.1.2.1 DEFAULT_DB

ExampleWrapper.**DEFAULT_DB** = ':memory:'

2.9.1.1.2.2 SCHEMAS

`ExampleWrapper.SCHEMAS = [('users', 'CREATE TABLE users (id INTEGER PRIMARY KEY AUTOINCREMENT`

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

<code>setUp()</code>	Hook method for setting up the test fixture before exercising it.
<code>tearDown()</code>	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 = '/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 = '/home/docs/.privex_sqlite'

If an absolute path isn't given, store the sqlite3 database file in this folder

DEFAULT_DB_NAME = '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 = 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

<i>BasePostgresTest</i> ([methodName])	Shared base class for Postgres tests.
<i>ExamplePostgresWrapper</i> (*args, **kwargs)	A wrapper around <i>PostgresWrapper</i> and <i>TestWrapperMixin</i> for use in Postgres tests.
<i>TestPostgresBuilder</i> ([methodName])	
<i>TestPostgresWrapper</i> ([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 finishes to avoid leftover tables/rows

wrp = None

Holds a *ExamplePostgresWrapper* instance for use by test cases

2.9.2.1.1 Methods

Methods

<i>setUp()</i>	Deletes and recreates all tables to avoid leftover tables/rows
<i>setUpClass()</i>	Sets up a <i>ExamplePostgresWrapper</i> instance under <i>wrp</i> for use by tests
<i>tearDown()</i>	Deletes all tables after each test finishes 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 finishes to avoid leftover tables/rows

2.9.2.1.2 Attributes

Attributes

<i>conn</i>	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

<code>__init__(*args, **kwargs)</code>	Initialise the database wrapper class.
<code>find_user(id)</code>	
<code>insert_user(first_name, last_name)</code>	

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 = 'privex_py_db'`

2.9.2.2.2 SCHEMAS

```
ExamplePostgresWrapper.SCHEMAS = [('users', 'CREATE TABLE users (id SERIAL PRIMARY KEY, fi
```

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

<code>test_all_call()</code>	Insert two users, then verify they're returned from an <code>.all()</code> call using <i>PostgresQueryBuilder</i>
<code>test_generator_builder()</code>	Test obtaining PostgresBuilder results by calling <code>next()</code> on the builder object (like a generator)
<code>test_group_call()</code>	Insert 5 users with 3 "John"s, then run a select+where+group_by query and confirm COUNT returns 3 johns
<code>test_index_builder()</code>	Test obtaining PostgresBuilder results by accessing an index of the builder object
<code>test_iterate_builder()</code>	Test obtaining PostgresBuilder results by iterating over the builder object itself with a for loop
<code>test_query_all()</code>	Build a select all query using <i>PostgresQueryBuilder</i> and confirm the built query looks correct
<code>test_query_select_col_where()</code>	Build a query selecting specific columns plus a 'where AND' clause using <i>PostgresQueryBuilder</i> and confirm the built query looks correct
<code>test_query_select_col_where_group()</code>	Build a complex select+where+group_by query using <i>PostgresQueryBuilder</i> and confirm the built query looks correct
<code>test_query_select_col_where_order()</code>	Build a query selecting specific columns, a 'where AND' clause, and an 'ORDER BY' clause using <i>PostgresQueryBuilder</i> and confirm the built query looks correct
<code>test_query_where_first_name_last_name()</code>	Build a 'where AND' query using <i>PostgresQueryBuilder</i> and confirm the built query looks correct
<code>test_where_call()</code>	Insert three users, then retrieve Dave using <code>.where()</code> + <code>.fetch()</code> call on <i>PostgresQueryBuilder</i>

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 finishes to avoid leftover tables/rows

wrp = 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

<i>test_all_call()</i>	
<i>test_generator_builder()</i>	Test obtaining SQLiteQueryBuilder results by calling <i>next()</i> on the builder object (like a generator)
<i>test_group_call()</i>	

Continued on next page

Table 47 – continued from previous page

<code>test_index_builder()</code>	Test obtaining SqliteQueryBuilder results by accessing an index of the builder object
<code>test_iterate_builder()</code>	Test obtaining SqliteQueryBuilder results by iterating over the builder object itself with a for loop
<code>test_query_all()</code>	
<code>test_query_select_col_where()</code>	
<code>test_query_select_col_where_group()</code>	
<code>test_query_select_col_where_order()</code>	
<code>test_query_where_first_name_last_name()</code>	
<code>test_where_call()</code>	

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')
```


INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

p

- `privex.db.base`, 9
- `privex.db.postgres`, 35
- `privex.db.query.base`, 60
- `privex.db.query.postgres`, 71
- `privex.db.query.sqlite`, 78
- `privex.db.sqlite`, 47
- `privex.db.types`, 56

t

- `tests`, 82
- `tests.base`, 85
- `tests.test_postgres`, 93
- `tests.test_sqlite_builder`, 103
- `tests.test_sqlite_wrapper`, 106

Symbols

`__init__()` (*privex.db.base.CursorManager* method), 11
`__init__()` (*privex.db.base.GenericDBWrapper* method), 20
`__init__()` (*privex.db.postgres.PostgresWrapper* method), 39
`__init__()` (*privex.db.query.base.BaseQueryBuilder* method), 63
`__init__()` (*privex.db.query.postgres.PostgresQueryBuilder* method), 74
`__init__()` (*privex.db.sqlite.SQLiteWrapper* method), 50
`__init__()` (*privex.db.types.GenericConnection* method), 57
`__init__()` (*privex.db.types.GenericCursor* method), 58
`__init__()` (*tests.base.ExampleWrapper* method), 86
`__init__()` (*tests.test_postgres.ExamplePostgresWrapper* method), 96

A

`action()` (*privex.db.base.GenericDBWrapper* method), 14, 20, 29
`all()` (*privex.db.query.base.BaseQueryBuilder* method), 60, 63, 69
`all()` (*privex.db.query.postgres.PostgresQueryBuilder* method), 72, 74, 77
`all()` (*privex.db.query.sqlite.SQLiteQueryBuilder* method), 79–81
`all()` (*tests.base.BaseQueryBuilder* method), 90
`all()` (*tests.base.SQLiteQueryBuilder* method), 92
`AsyncCursorManager` (class in *privex.db.base*), 28
`AUTO_ZIP_COLS` (*privex.db.base.GenericDBWrapper* attribute), 13, 26, 29
`AUTO_ZIP_COLS` (*privex.db.postgres.PostgresWrapper* attribute), 43

B

`BasePostgresTest` (class in *tests.test_postgres*), 93, 102

`BaseQueryBuilder` (class in *privex.db.query.base*), 60, 68
`BaseQueryBuilder` (class in *tests.base*), 90
`build_query()` (*privex.db.query.base.BaseQueryBuilder* method), 60, 64, 69
`build_query()` (*privex.db.query.postgres.PostgresQueryBuilder* method), 72, 74, 77
`build_query()` (*privex.db.query.sqlite.SQLiteQueryBuilder* method), 79–81
`build_query()` (*tests.base.BaseQueryBuilder* method), 90
`build_query()` (*tests.base.SQLiteQueryBuilder* method), 92
`builder()` (*privex.db.postgres.PostgresWrapper* method), 40
`builder()` (*privex.db.sqlite.SQLiteWrapper* method), 50

C

`can_cleanup` (*privex.db.base.AsyncCursorManager* attribute), 28
`can_cleanup` (*privex.db.base.CursorManager* attribute), 10, 28
`close()` (*privex.db.base.CursorManager* method), 11
`close()` (*privex.db.types.GenericConnection* method), 57
`close()` (*privex.db.types.GenericCursor* method), 59
`close_cursor()` (*privex.db.base.GenericDBWrapper* method), 21
`close_cursor()` (*privex.db.query.base.BaseQueryBuilder* method), 64
`commit()` (*privex.db.types.GenericConnection* method), 58
`conn` (*privex.db.base.GenericAsyncDBWrapper* attribute), 28
`conn` (*privex.db.sqlite.SQLiteAsyncWrapper* attribute), 53
`conn` (*tests.test_postgres.BasePostgresTest* attribute), 93, 95, 102
`conn()` (*privex.db.base.GenericDBWrapper* property), 14, 27, 30
`conn()` (*privex.db.postgres.PostgresWrapper* property), 36, 43, 44

`conn()` (*privex.db.query.postgres.PostgresQueryBuilder* property), 76

`conn()` (*privex.db.query.sqlite.SQLiteQueryBuilder* property), 81

`conn()` (*privex.db.sqlite.SQLiteWrapper* property), 49, 51, 55

`conn()` (*tests.base.SQLiteWrapper* property), 89

`connection` (*privex.db.query.base.BaseQueryBuilder* attribute), 68

`connection` (*privex.db.query.sqlite.SQLiteQueryBuilder* attribute), 81

`create_schema()` (*privex.db.base.GenericDBWrapper* method), 14, 21, 30

`create_schemas()` (*privex.db.base.GenericDBWrapper* method), 14, 21, 30

`cursor()` (*privex.db.base.GenericDBWrapper* property), 28

`cursor()` (*privex.db.query.base.BaseQueryBuilder* property), 68

`cursor()` (*privex.db.query.postgres.PostgresQueryBuilder* property), 76

`cursor()` (*privex.db.types.GenericConnection* method), 58

`cursor_cls()` (*privex.db.postgres.PostgresWrapper* property), 44

`cursor_map` (*privex.db.postgres.PostgresWrapper* attribute), 44

`cursor_to_dict()` (in module *privex.db.base*), 35

`CursorManager` (class in *privex.db.base*), 10, 28

D

`db` (*privex.db.postgres.PostgresWrapper* attribute), 36, 44

`db` (*privex.db.sqlite.SQLiteAsyncWrapper* attribute), 53

`db` (*privex.db.sqlite.SQLiteWrapper* attribute), 49, 55

`db` (*tests.base.SQLiteWrapper* attribute), 89

`DBExecution` (class in *privex.db.base*), 12, 28

`DEFAULT` (*privex.db.query.base.QueryMode* attribute), 68

`DEFAULT_DB` (*privex.db.postgres.PostgresWrapper* attribute), 43

`DEFAULT_DB` (*privex.db.sqlite.SQLiteAsyncWrapper* attribute), 53

`DEFAULT_DB` (*privex.db.sqlite.SQLiteWrapper* attribute), 49, 51, 55

`DEFAULT_DB` (*tests.base.ExampleWrapper* attribute), 86

`DEFAULT_DB` (*tests.base.SQLiteWrapper* attribute), 89

`DEFAULT_DB` (*tests.test_postgres.ExamplePostgresWrapper* attribute), 96

`DEFAULT_DB_FOLDER` (*privex.db.sqlite.SQLiteAsyncWrapper* attribute), 53

`DEFAULT_DB_FOLDER` (*privex.db.sqlite.SQLiteWrapper* attribute), 49, 51, 55

`DEFAULT_DB_FOLDER` (*tests.base.SQLiteWrapper* attribute), 89

`DEFAULT_DB_NAME` (*privex.db.sqlite.SQLiteAsyncWrapper* attribute), 53

`DEFAULT_DB_NAME` (*privex.db.sqlite.SQLiteWrapper* attribute), 49, 51, 55

`DEFAULT_DB_NAME` (*tests.base.SQLiteWrapper* attribute), 89

`DEFAULT_ENABLE_EXECUTION_LOG` (*privex.db.base.GenericDBWrapper* attribute), 27

`DEFAULT_PLACEHOLDER` (*privex.db.base.GenericDBWrapper* attribute), 13, 29

`DEFAULT_QUERY_MODE` (*privex.db.base.GenericDBWrapper* attribute), 13, 27, 29

`DEFAULT_QUERY_MODE` (*privex.db.postgres.PostgresWrapper* attribute), 43

`DEFAULT_TABLE_LIST_QUERY` (*privex.db.base.GenericDBWrapper* attribute), 27

`DEFAULT_TABLE_LIST_QUERY` (*privex.db.postgres.PostgresWrapper* attribute), 43

`DEFAULT_TABLE_LIST_QUERY` (*privex.db.sqlite.SQLiteWrapper* attribute), 51

`DEFAULT_TABLE_QUERY` (*privex.db.base.GenericDBWrapper* attribute), 27

`DEFAULT_TABLE_QUERY` (*privex.db.postgres.PostgresWrapper* attribute), 43

`DEFAULT_TABLE_QUERY` (*privex.db.sqlite.SQLiteWrapper* attribute), 51

`description()` (*privex.db.base.CursorManager* property), 12

`drop_schemas()` (*privex.db.base.GenericDBWrapper* method), 15, 21, 30

`drop_table()` (*privex.db.base.GenericAsyncDBWrapper* method), 28

`drop_table()` (*privex.db.base.GenericDBWrapper* method), 15, 22, 30

`drop_table()` (*privex.db.postgres.PostgresWrapper* method), 36, 40, 44

`drop_tables()` (*privex.db.base.GenericDBWrapper* method), 15, 22, 31

E

ExampleAsyncWrapper (class in tests.base), 93
 ExamplePostgresWrapper (class in tests.test_postgres), 95, 102
 ExampleWrapper (class in tests.base), 85, 93
 execute() (privex.db.base.CursorManager method), 11
 execute() (privex.db.query.base.BaseQueryBuilder method), 64
 execute() (privex.db.types.GenericCursor method), 59
 executemany() (privex.db.types.GenericCursor method), 59

F

fetch() (privex.db.base.GenericDBWrapper method), 15, 22, 31
 fetch() (privex.db.query.base.BaseQueryBuilder method), 61, 64, 69
 fetch() (privex.db.query.postgres.PostgresQueryBuilder method), 72, 74, 77
 fetch() (privex.db.query.sqlite.SQLiteQueryBuilder method), 79–81
 fetch() (tests.base.BaseQueryBuilder method), 90
 fetch() (tests.base.SQLiteQueryBuilder method), 92
 fetch_next() (privex.db.query.base.BaseQueryBuilder method), 61, 64, 69
 fetch_next() (privex.db.query.postgres.PostgresQueryBuilder method), 73, 75, 78
 fetch_next() (privex.db.query.sqlite.SQLiteQueryBuilder method), 79, 80, 82
 fetch_next() (tests.base.BaseQueryBuilder method), 91
 fetch_next() (tests.base.SQLiteQueryBuilder method), 93
 fetchall() (privex.db.base.CursorManager method), 11
 fetchall() (privex.db.base.GenericDBWrapper method), 15, 23, 31
 fetchall() (privex.db.types.GenericCursor method), 59
 fetchmany() (privex.db.base.CursorManager method), 11
 fetchmany() (privex.db.types.GenericCursor method), 59
 fetchone() (privex.db.base.CursorManager method), 11
 fetchone() (privex.db.base.GenericDBWrapper method), 15, 23, 31
 fetchone() (privex.db.types.GenericCursor method), 59
 find_user() (tests.test_postgres.ExamplePostgresWrapper method), 96

G

GenericAsyncConnection (class in privex.db.types), 59
 GenericAsyncCursor (class in privex.db.types), 59
 GenericAsyncDBWrapper (class in privex.db.base), 28
 GenericConnection (class in privex.db.types), 57, 59
 GenericCursor (class in privex.db.types), 58, 59
 GenericDBWrapper (class in privex.db.base), 12, 29
 get_cursor() (privex.db.base.GenericAsyncDBWrapper method), 28
 get_cursor() (privex.db.base.GenericDBWrapper method), 15, 23, 31
 get_cursor() (privex.db.query.base.BaseQueryBuilder method), 61, 64, 69
 get_cursor() (privex.db.query.postgres.PostgresQueryBuilder method), 73, 75, 78
 get_cursor() (privex.db.query.sqlite.SQLiteAsyncWrapper method), 53
 get_cursor() (tests.base.BaseQueryBuilder method), 91
 group_by() (privex.db.query.base.BaseQueryBuilder method), 61, 65, 69
 group_by() (tests.base.BaseQueryBuilder method), 91

I

insert() (privex.db.base.GenericDBWrapper method), 16, 32
 insert() (privex.db.postgres.PostgresWrapper method), 36, 44
 insert() (privex.db.sqlite.SQLiteAsyncWrapper method), 54
 insert() (privex.db.sqlite.SQLiteWrapper method), 49, 55
 insert() (tests.base.SQLiteWrapper method), 89
 insert_user() (tests.test_postgres.ExamplePostgresWrapper method), 96
 is_context_manager (privex.db.base.AsyncCursorManager attribute), 28
 is_context_manager (privex.db.base.CursorManager attribute), 10, 28

L

last_insert_id() (privex.db.postgres.PostgresWrapper method), 37, 40, 45
 lastrowid() (privex.db.base.CursorManager property), 12
 limit() (privex.db.query.base.BaseQueryBuilder method), 61, 65, 70
 limit() (tests.base.BaseQueryBuilder method), 91

`list_tables()` (*privex.db.base.GenericDBWrapper method*), 16, 23, 32

`list_tables()` (*privex.db.postgres.PostgresWrapper method*), 37, 40, 45

M

`make_connection()`
(*privex.db.base.GenericDBWrapper method*), 16, 24, 32

O

`order()` (*privex.db.query.base.BaseQueryBuilder method*), 61, 65, 70

`order()` (*tests.base.BaseQueryBuilder method*), 91

`order_by()` (*privex.db.query.base.BaseQueryBuilder method*), 61, 65, 70

`order_by()` (*tests.base.BaseQueryBuilder method*), 91

P

`PostgresQueryBuilder` (*class in privex.db.query.postgres*), 71, 76

`PostgresWrapper` (*class in privex.db.postgres*), 35, 44

`privex.db.base` (*module*), 9

`privex.db.postgres` (*module*), 35

`privex.db.query.base` (*module*), 60

`privex.db.query.postgres` (*module*), 71

`privex.db.query.sqlite` (*module*), 78

`privex.db.sqlite` (*module*), 47

`privex.db.types` (*module*), 56

`PrivexDBTestBase` (*class in tests.base*), 87, 88

`pytestmark` (*tests.test_postgres.TestPostgresBuilder attribute*), 100

`pytestmark` (*tests.test_postgres.TestPostgresWrapper attribute*), 102

Q

`Q_DEFAULT_PLACEHOLDER`
(*privex.db.query.base.BaseQueryBuilder attribute*), 67

`Q_DEFAULT_PLACEHOLDER`
(*privex.db.query.postgres.PostgresQueryBuilder attribute*), 76

`Q_DEFAULT_PLACEHOLDER`
(*privex.db.query.sqlite.SqliteQueryBuilder attribute*), 81

`Q_GROUP_BY_CLAUSE`
(*privex.db.query.base.BaseQueryBuilder attribute*), 67

`Q_LIMIT_CLAUSE` (*privex.db.query.base.BaseQueryBuilder attribute*), 67

`Q_OFFSET_CLAUSE` (*privex.db.query.base.BaseQueryBuilder attribute*), 67

`Q_ORDER_CLAUSE` (*privex.db.query.base.BaseQueryBuilder attribute*), 67

`Q_POST_QUERY` (*privex.db.query.base.BaseQueryBuilder attribute*), 67

`Q_PRE_QUERY` (*privex.db.query.base.BaseQueryBuilder attribute*), 67

`Q_PRE_QUERY` (*privex.db.query.postgres.PostgresQueryBuilder attribute*), 76

`Q_PRE_QUERY` (*privex.db.query.sqlite.SqliteQueryBuilder attribute*), 81

`Q_SELECT_CLAUSE` (*privex.db.query.base.BaseQueryBuilder attribute*), 67

`Q_WHERE_CLAUSE` (*privex.db.query.base.BaseQueryBuilder attribute*), 67

`query()` (*privex.db.base.GenericDBWrapper method*), 17, 24, 32

`query()` (*privex.db.postgres.PostgresWrapper method*), 37, 41, 45

`query_mode` (*privex.db.base.GenericDBWrapper attribute*), 18, 34

`query_mode_cursor()`
(*privex.db.query.postgres.PostgresQueryBuilder method*), 73, 75, 78

`QueryMode` (*class in privex.db.query.base*), 68, 71

`QueryMode` (*class in tests.base*), 93

R

`recreate_schemas()`
(*privex.db.base.GenericDBWrapper method*), 18, 25, 34

`rollback()` (*privex.db.types.GenericConnection method*), 58

`ROW_DICT` (*privex.db.query.base.QueryMode attribute*), 68

`ROW_TUPLE` (*privex.db.query.base.QueryMode attribute*), 68

`rowcount()` (*privex.db.base.CursorManager property*), 12

S

`SCHEMAS` (*privex.db.base.GenericDBWrapper attribute*), 13, 27, 29

`SCHEMAS` (*tests.base.ExampleWrapper attribute*), 87

`SCHEMAS` (*tests.test_postgres.ExamplePostgresWrapper attribute*), 97

`select()` (*privex.db.query.base.BaseQueryBuilder method*), 61, 65, 70

`select()` (*tests.base.BaseQueryBuilder method*), 91

`select_date()` (*privex.db.query.postgres.PostgresQueryBuilder method*), 73, 75, 78

`setUp()` (*tests.base.PrivexDBTestBase method*), 87, 88

`setUp()` (*tests.test_postgres.BasePostgresTest method*), 93, 94, 102

setUpClass() (*tests.test_postgres.BasePostgresTestClass method*), 93, 94, 102
 SqliteAsyncWrapper (*class in privex.db.sqlite*), 51
 SqliteQueryBuilder (*class in privex.db.query.sqlite*), 79, 81
 SqliteQueryBuilder (*class in tests.base*), 92
 SqliteWrapper (*class in privex.db.sqlite*), 48, 54
 SqliteWrapper (*class in tests.base*), 88

T

table_exists() (*privex.db.base.GenericDBWrapper method*), 18, 26, 34
 table_exists() (*privex.db.postgres.PostgresWrapper method*), 39, 42, 47
 tables_created (*privex.db.base.GenericDBWrapper attribute*), 19, 28, 34
 tearDown() (*tests.base.PrivexDBTestBase method*), 87, 88
 tearDown() (*tests.test_postgres.BasePostgresTestClass method*), 93, 94, 102
 test_action_update() (*tests.test_postgres.TestPostgresWrapper method*), 101
 test_action_update() (*tests.test_sqlite_wrapper.TestSQLiteWrapper method*), 106
 test_all_call() (*tests.test_postgres.TestPostgresBuilder method*), 97, 98, 102
 test_all_call() (*tests.test_sqlite_builder.TestSQLiteBuilder method*), 104
 test_find_user_dict_mode() (*tests.test_postgres.TestPostgresWrapper method*), 101
 test_find_user_dict_mode() (*tests.test_sqlite_wrapper.TestSQLiteWrapper method*), 107
 test_find_user_nonexistent() (*tests.test_postgres.TestPostgresWrapper method*), 101
 test_find_user_nonexistent() (*tests.test_sqlite_wrapper.TestSQLiteWrapper method*), 107
 test_generator_builder() (*tests.test_postgres.TestPostgresBuilder method*), 97, 99, 102
 test_generator_builder() (*tests.test_sqlite_builder.TestSQLiteBuilder method*), 103–105
 test_get_users_dict() (*tests.test_postgres.TestPostgresWrapper method*), 101
 test_get_users_dict() (*tests.test_sqlite_wrapper.TestSQLiteWrapper method*), 107
 test_get_users_tuple() (*tests.test_postgres.TestPostgresWrapper method*), 101
 test_get_users_tuple() (*tests.test_sqlite_wrapper.TestSQLiteWrapper method*), 107
 test_group_call() (*tests.test_postgres.TestPostgresBuilder method*), 97, 99, 102
 test_group_call() (*tests.test_sqlite_builder.TestSQLiteBuilder method*), 104
 test_index_builder() (*tests.test_postgres.TestPostgresBuilder method*), 97, 99, 102
 test_index_builder() (*tests.test_sqlite_builder.TestSQLiteBuilder method*), 103–105
 test_insert_find_user() (*tests.test_postgres.TestPostgresWrapper method*), 101
 test_insert_find_user() (*tests.test_sqlite_wrapper.TestSQLiteWrapper method*), 107
 test_iterate_builder() (*tests.test_postgres.TestPostgresBuilder method*), 97, 99, 102
 test_iterate_builder() (*tests.test_sqlite_builder.TestSQLiteBuilder method*), 103–105
 test_query_all() (*tests.test_postgres.TestPostgresBuilder method*), 97, 99, 102
 test_query_all() (*tests.test_sqlite_builder.TestSQLiteBuilder method*), 104
 test_query_select_col_where() (*tests.test_postgres.TestPostgresBuilder method*), 97, 99, 102
 test_query_select_col_where() (*tests.test_sqlite_builder.TestSQLiteBuilder method*), 105
 test_query_select_col_where_group() (*tests.test_postgres.TestPostgresBuilder method*), 97, 99, 102
 test_query_select_col_where_group() (*tests.test_sqlite_builder.TestSQLiteBuilder method*), 105
 test_query_select_col_where_order() (*tests.test_postgres.TestPostgresBuilder method*), 97, 99, 103
 test_query_select_col_where_order() (*tests.test_sqlite_builder.TestSQLiteBuilder method*), 105
 test_query_where_first_name_last_name() (*tests.test_postgres.TestPostgresBuilder method*), 103

method), 97, 100, 103
test_query_where_first_name_last_name()
 (*tests.test_sqlite_builder.TestSQLiteBuilder*
 method), 105
test_tables_created()
 (*tests.test_postgres.TestPostgresWrapper*
 method), 101
test_tables_created()
 (*tests.test_sqlite_wrapper.TestSQLiteWrapper*
 method), 107
test_tables_drop()
 (*tests.test_postgres.TestPostgresWrapper*
 method), 101
test_tables_drop()
 (*tests.test_sqlite_wrapper.TestSQLiteWrapper*
 method), 107
test_where_call()
 (*tests.test_postgres.TestPostgresBuilder*
 method), 97, 100, 103
test_where_call()
 (*tests.test_sqlite_builder.TestSQLiteBuilder*
 method), 105
TestPostgresBuilder (*class in tests.test_postgres*),
 97, 102
TestPostgresWrapper (*class in tests.test_postgres*),
 100, 103
tests (*module*), 82
tests.base (*module*), 85
tests.test_postgres (*module*), 93
tests.test_sqlite_builder (*module*), 103
tests.test_sqlite_wrapper (*module*), 106
TestSQLiteBuilder (*class in*
 tests.test_sqlite_builder), 103, 105
TestSQLiteWrapper (*class in*
 tests.test_sqlite_wrapper), 106, 107

U

User (*class in tests.base*), 88, 93

W

where() (*privex.db.query.base.BaseQueryBuilder*
 method), 62, 66, 70
where() (*tests.base.BaseQueryBuilder method*), 91
where_or() (*privex.db.query.base.BaseQueryBuilder*
 method), 62, 66, 70
where_or() (*tests.base.BaseQueryBuilder method*),
 92
wrp (*tests.test_postgres.BasePostgresTest attribute*), 93,
 102