# pytest$_{pgsql}$Documentation

***Release 1.0.2***

**Clover Health**

**Dec 06, 2017**

# Contents

CHAPTER 1

# Clean PostgreSQL Databases for Your Tests

## 1.1 What is `pytest-pgsql`?

`pytest-pgsql` is a pytest plugin you can use to write unit tests that utilize a temporary PostgreSQL database that gets cleaned up automatically after every test runs, allowing each test to run on a completely clean database (with *limitations*).

The plugin gives you two fixtures you can use in your tests: `postgresql_db` and `transacted_postgresql_db`. Both of these give you similar interfaces to access to the database, but have slightly different use cases (see below).

## 1.2 General Usage

You can use a session, connection, or engine - the choice is up to you. `postgresql_db` and `transacted_postgresql_db` both give you a session, but `postgresql_db` exposes its engine and `transacted_postgresql_db` exposes its connection:

```
def test_orm(postgresql_db):
    instance = Person(name='Foo Bar')
    postgresql_db.session.add(instance)
    postgresql_db.session.commit()
    with postgresql_db.engine.connect() as conn:
        do_thing(conn)

def test_connection(transacted_postgresql_db):
    instance = Person(name='Foo Bar')
    transacted_postgresql_db.session.add(instance)
    transacted_postgresql_db.session.commit()

    transacted_postgresql_db.connection.execute('DROP TABLE my_table')
```

### 1.2.1 Which Do I Use?

There are a few differences between the transacted and non-transacted versions of the PostgreSQL database fixture. They are as follows:

**Transacted is faster**

The major advantage of `transacted_postgresql_db` is that resetting the database to its original state is much faster than `postgresql_db`. Unfortunately, that comes at the expense of having to run the entire test in a single transaction. This means you can't call execute a `COMMIT` statement anywhere in your tests, or you'll risk causing nondeterministic bugs in your tests and possibly hiding bugs your production code.

For a full description of what can go wrong if you execute `COMMIT` and how to get around this limitation, see the *Tips* section.

**Non-transacted is more flexible**

`postgresql_db` is more flexible than its transacted counterpart because it doesn't have to run in a single transaction, but teardown is more time-consuming because every single table, schema, extension, etc. needs to be manually reset. Tests that can't run in one transaction (e.g. data needs to be shared across threads) must use the `postgresql_db` fixture.

### 1.2.2 Limitations

It's important to note that at the moment the fixtures *can't* revert some changes if a top-level commit[1] has been executed. As far as we know this only applies to objects (extensions, schemas, tables, etc.) that existed before the test started.

The following is a non-exhaustive list of changes that cannot be reverted after a top-level commit:

- Modifications to the structure of preexisting tables, including
  - Added/removed/modified rows
  - Schema changes, e.g. `ALTER TABLE`, `ALTER COLUMN`, etc.) Tables that were renamed or moved to different schemas will be moved back.
  - Added/removed/modified constraints or indexes
- Schemas, tables, and other objects that were dropped during the test cannot be fully restored. Schemas can be recreated but may have lost some of their original contents.
- Database settings such as changes to the search path won't be reverted to defaults.
- Ownership and permission changes will persist until the end of the test session.

## 1.3 Utility Functions

There are a few utility functions each fixture gives you as well. The following examples use `postgresql_db`, but `transacted_postgresql_db` behaves similarly.

### 1.3.1 Extensions

Since version 9.1 Postgres supports extensions. You can check for the presence of and install extensions like so:

[1] A *top-level commit* is a commit made on the outermost transaction of a session. SQLAlchemy allows you to nest transactions so that changes are only persisted to the database when the outermost one is committed. For more information, see Using SAVEPOINT in the SQLAlchemy docs.

```
>>> postgresql_db.is_extension_available('asdf')  # Can I install this extension?
False
>>> postgresql_db.is_extension_available('uuid-ossp')  # Maybe this one is supported..
↪.
True
>>> postgresql_db.install_extension('uuid-ossp')
True
>>> postgresql_db.is_extension_installed('uuid-ossp')
True
```

`install_extension` has additional arguments to allow control over which schema the extension is installed in, what to do if the extension is already installed, and so on. See the documentation for descriptions of these features.

### 1.3.2 Schemas

You can create table schemas by calling `create_schema` like so:

```
postgresql_db.create_schema('foo')
```

The function will throw an exception if the schema already exists. If you only want to create the schema if it doesn't already exist, pass `True` for the `exists_ok` argument:

```
postgresql_db.create_schema('foo', exists_ok=True)
```

To quickly see if a table schema exists, call `has_schema`:

```
>>> postgresql_db.has_schema('public')
True
```

Note that multiple schemas can be created at once:

```
postgresql_db.create_schema('foo', 'bar')
```

### 1.3.3 Tables

Similarly, you can create tables in the database with `create_table`. You can pass `sqlalchemy.Table` instances or ORM declarative model classes:

```
# Just a regular Table.
my_table = Table('abc', MetaData(), Column('def', Integer()))

# A declarative model works too.
class MyORMModel(declarative_base()):
    id = Column(Integer, primary_key=True)

# Pass a variable amount of tables to create
postgresql_db.create_table(my_table, MyORMModel)
```

There are several ways to check to see if a table exists:

```
>>> postgresql_db.has_table('mytable')    # 'mytable' in *any* schema
True

>>> postgresql_db.has_table('the_schema.the_table')    # 'the_table' only in 'the_
↪schema'
```

```
False

>>> table = Table('foo', MetaData(), Column('bar', Integer()))
>>> postgresql_db.has_table(table)
False
>>> postgresql_db.create_table(table)
>>> postgresql_db.has_table(table)
True

>>> postgresql_db.has_table(MyORMModelClass)
True
```

### 1.3.4 Manipulating Time

Both database fixtures use freezegun to allow you to freeze time inside a block of code. You can use it in a variety of ways:

As a context manager:

```
with postgresql.time.freeze('December 31st 1999 11:59:59 PM') as freezer:
    # Time is frozen inside the database *and* Python.
    now = postgresql_db.session.execute('SELECT NOW()').scalar()
    assert now.date() == datetime.date(1999, 12, 31)
    assert datetime.date.today() == datetime.date(1999, 12, 31)

    # Advance time by 1 second so we roll over into the new year
    freezer.tick()

    now = postgresql_db.session.execute('SELECT NOW()').scalar()
    assert now.date() == datetime.date(2000, 1, 1)
```

Manually calling the `freeze()` and `unfreeze()` functions:

```
postgresql_db.time.freeze(datetime.datetime(1999, 12, 31, 23, 59, 59))
...
postgresql_db.time.unfreeze()
```

You can also freeze time for an entire test if you like using the `freeze_time` decorator:

```
@pytest_pgsql.freeze_time(datetime.datetime(2038, 1, 19, 3, 14, 7))
def test_freezing(postgresql_db):
    today = postgresql_db.session.execute(
        "SELECT EXTRACT('YEAR' FROM CURRENT_DATE)").scalar()
    assert today.year == 2038
    assert datetime.date.today() == datetime.date(2038, 1, 19)
```

If you choose not to use the context manager but still need control over the flow of time, the `FrozenDateTimeFactory` instance can be accessed with the `freezer` attribute:

```
postgresql_db.time.freeze(datetime.datetime(1999, 12, 31, 23, 59, 59))
postgresql_db.time.freezer.tick()

now = postgresql_db.session.execute('SELECT LOCALTIME').scalar()
assert now == datetime.datetime(2000, 1, 1)

postgresql_db.time.unfreeze()
```

See the documentation for `SQLAlchemyFreezegun` for detailed information on what this feature can and can't do for you.

### 1.3.5 General-Purpose Functions

`postgresql_db` and `transacted_postgresql_db` provide some general-purpose functions to ease test setup and execution.

- `load_csv` loads a CSV file into an existing table.
- `run_sql_file` executes a SQL script, optionally performing variable binding.

## 1.4 Fixture Customization

You may find that the default settings for the database fixtures are inadequate for your needs. You can customize how the engine and database fixtures are created with the use of facilities provided in the `ext` ("extension") module.

### 1.4.1 Customizing the Engine

Suppose we want our database engine to transparently encode a `datetime` or `decimal.Decimal` object in JSON for us. We can create our own engine that'll do so by using `create_engine_fixture()`:

```
import pytest_pgsql
import simplejson as json

json_engine = pytest_pgsql.create_engine_fixture(
    'json_engine', json_serializer=json.dumps, json_deserializer=json.loads)
```

Great! Now we have a database engine that can encode and decode timestamps and fixed-point decimals without any manual conversion on our part. This is not the only way we can customize the engine–you can pass any keyword argument to `create_engine_fixture()` that's valid for `sqlalchemy.create_engine`. See the documentation there for a full list of what you can do.

So how do we use it with all the benefits we get from `postgresql_db` and `transacted_postgresql_db`?

### 1.4.2 Customizing Database Fixtures

You can create your own database fixture by choosing any subclass of `PostgreSQLTestDBBase` and invoking its `create_fixture()` method, passing the name of your new fixture and the name of the engine fixture to use:

```
import pytest_pgsql

simplejson_db = pytest_pgsql.PostgreSQLTestDB.create_fixture(
    'simplejson_db', 'json_engine')
```

We now have a function-scoped database fixture identical to `postgresql_db` but with more comprehensive JSON serialization! If I wanted a faster transacted version, I could use `TransactedPostgreSQLTestDB` as the base class instead:

```
import pytest_pgsql

tsimplejson_db = pytest_pgsql.TransactedPostgreSQLTestDB.create_fixture(
    'tsimplejson_db', 'json_engine')
```

You can change how the fixture is created by passing any keyword arguments that are valid for the `pytest.fixture` decorator. For example, you can set the scope of the fixture to the module level by using the `scope` keyword argument:

```
simplejson_db = pytest_pgsql.PostgreSQLTestDB.create_fixture(
    'simplejson_db', 'json_engine', scope='module')
```

Now, in our tests we can use the fixtures directly:

```
import datetime
import sqlalchemy as sqla
import sqlalchemy.dialects.postgresql as sqla_pg

def test_blah(simplejson_db):
    meta = sqla.MetaData(bind=simplejson_db.connection)
    table = sqla.Table('test', meta, sqla.Column('col', sqla_pg.JSON))
    meta.create_all()

    simplejson_db.connection.execute(table.insert(), [
        {'col': datetime.datetime.now()}
    ])
```

## 1.5 Command Line Options

### 1.5.1 --pg-driver

Use this to tell the fixtures which database driver to use. The default if not given is psycopg2. SQLAlchemy supports all the drivers listed here but some may not work with `pytest_pgsql`.[2]

Usage:

```
pytest --pg-driver pygresql
```

### 1.5.2 --pg-extensions

If many of your tests are going to need one or more particular extensions, you can tell `pytest_pgsql` to install them at the beginning of the test session. This is *much* faster and more efficient than calling `install_extension` for each test.

Pass a comma-separated list of the extensions you need on the command line like so:

```
# Install "uuid-ossp" and "pg_tgrm" so all tests can use it
pytest --pg-extensions=uuid-ossp,pg_tgrm
```

### 1.5.3 --pg-work-mem

The `--pg-work-mem` option allows you to tweak the amount of memory that sort operations can use. The Postgres default is rather low (4MB at time of writing) so `pytest_pgsql` uses 32MB as its default. Try adjusting this value up or down to find the optimal value for your test suite, or use `0` to use the server default.

[2] pg8000 is one such driver that doesn't work. If your driver uses server-side prepared statements instead of doing the parametrization in Python, your driver *will not* work. This is because PostgreSQL's prepared statements don't support parametrizing `IN` clauses, something currently required by `is_dirty()`.

```
# Increase the amount of working memory to 64MB
pytest --pg-work-mem=64

# Disable tweaking and use the server default
pytest --pg-work-mem=0
```

For more information:

- PostgreSQL documentation: Resource Consumption
- PostgreSQL wiki: Tuning your PostgreSQL Server

## 1.6 Tips

### 1.6.1 Be careful with `COMMIT`

When using `transacted_postgresql_db`, do *not* use `connection.execute()` to commit changes made:

```
# This is fine...
transacted_postgresql_db.session.commit()

# So is this...
transaction = transacted_postgresql_db.connection.begin()
transaction.commit()

# But this is not.
transacted_postgresql_db.connection.execute('COMMIT')
```

The problem with executing `COMMIT` in a transacted PostgreSQL testcase is that all tests assume they're running in a clean database. Committing persists changes made, so the database is no longer clean *for the rest of the session*. Let's see how that can be harmful:

1. Suppose we have a test A that creates some rows in table X and executes a `COMMIT`. We now have one row in X.
2. Test B creates another row in X. Now there are two rows in the table, but test B thinks there's only one.
3. Test B does a search for all rows in X that fit some criterion, but there's a bug in the code and it unintentionally skips the first row it finds. If test A created a row meeting that criterion, then test B will pass *even though there's a bug in B's code*.

Furthermore, this will only happen *if* test A runs before test B. Thus, adding or removing any tests can change the order and make the error appear and disappear seemingly at random.

CHAPTER 2

# Installation

## 2.1 System Requirements

- PostgreSQL 9.1 or greater must be installed.
- Python 3.3 or greater. Compatibility with PyPy3.5 is untested and not guaranteed.
- You must use `psycopg2` as your database driver.

**Note:** Due to the way that `tox` works with environment setup, if your system's Python 3 version is 3.6.x and you installed any Python package that uses `cli-helpers` version 0.2.0 or greater, `make setup` will fail. This is due to a known bug in `pbr` and as of 2017-12-02 there is no workaround that won't potentially break other packages.

## 2.2 Setup

```
pip3 install pytest-pgsql
```

# CHAPTER 3

# Release Notes

```
1.0.3
-----

* Fixed issue when running "make setup"

1.0.2
-----

* Fix Github link in package metadata and remove custom URL creation based on driver

1.0.1
-----

* Fixed README docs link
* sem-ver: api-break, Initial release
* Initial commit
```

CHAPTER 4

# Contributing Guide

## 4.1 Setup

Set up your development environment with:

```
git clone git@github.com:CloverHealth/pytest-pgsql.git
cd pytest-pgsql
make setup
```

`make setup` will setup a virtual environment managed by pyenv and install dependencies.

Note that if you'd like to use something else to manage dependencies other than pyenv, call `make dependencies` instead of `make setup`.

## 4.2 Testing and Validation

Run the tests with:

```
make test
```

Validate the code with:

```
make validate
```

## 4.3 Documentation

Sphinx documentation can be built with:

```
make docs
```

The static HTML files are stored in the `docs/_build/html` directory. A shortcut for opening them on OSX is:

```
make open_docs
```

## 4.4 Releases and Versioning

Anything that is merged into the master branch will be automatically deployed to PyPI. Documentation will be published to ReadTheDocs soon.

The following files will be generated and should *not* be edited by a user:

- `ChangeLog` - Contains the commit messages of the releases. Please have readable commit messages in the master branch and squash and merge commits when necessary.
- `AUTHORS` - Contains the contributing authors.
- `version.py` - Automatically updated to include the version string.

This project uses Semantic Versioning through PBR. This means when you make a commit, you can add a message like:

```
sem-ver: feature, Added this functionality that does blah.
```

Depending on the sem-ver tag, the version will be bumped in the right way when releasing the package. For more information, about PBR, go the the PBR docs.

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search