# asyncpgsa Documentation

*Release 0.1*

**Rajhans**

**May 21, 2019**

# Contents

A python library wrapper around asyncpg for use with sqlalchemy.

CHAPTER 1

# sqlalchemy ORM

Currently this repo does not support SA ORM, only SA Core.

As we at canopy do not use the ORM, if you would like to have ORM support feel free to PR it. You would need to create an "engine" interface, and that should be it. Then you can bind your sessions to the engine.

CHAPTER 2

# sqlalchemy Core

This repo supports sqlalchemy core.

install Currently this repo does not support SA ORM, only SA Core.

As we at canopy do not use the ORM, if you would like to have ORM support feel free to PR it. You would need to create an “engine” interface, and that should be it. Then you can bind your sessions to the engine.

CHAPTER 3

# Install

CHAPTER 4

# Examples

There are two ways to use this library, the first is by establishing a pool, and using that. The next is a singleton called PG. This manages a pool for you and makes it easy to call from module to module without having to pass anything around.

## 4.1 Query Object

all there examples use the variable `query`, this is a query object. It can either be a string or a sqlalchemy core statement. Here are some examples

```
# string
query = 'select * from sqrt(16)'

#sqlalchemy statement with table object
import sqlalchemy as sa
pg_tables = pg_tables = sa.Table(
         'pg_tables', sa.MetaData(),
         sa.Column('schemaname'),
         sa.Column('tablename'),
         sa.Column('tableowner'),
         sa.Column('tablespace'),
         sa.Column('hasindexes')
)
query = pg_tables.select().where(pg_tables.c.schemaname == 'pg_catalog')

# sqlalchemy statement with parameters
query = sa.select('*') \
    .select_from(sa.text('sqrt(:num) as a')) \
    .select_from(sa.text('sqrt(:a2) as b')) \
    .select_from(sa.text('sqrt(:z3) as c')) \
    .params(num=16, a2=36, z3=25)
```

## 4.2 PG Singleton

If you want the highest level of abstraction, you can can use the singleton object. This will create a pool for you.

### 4.2.1 Init

Before you can run any queries you first have to initialize the pool

```
await pg.init(
    host=HOST,
    port=PORT,
    database=DB_NAME,
    user=USER,
    # loop=loop,
    password=PASS,
    min_size=5,
    max_size=10
)
```

### 4.2.2 Query

Query is for making read only select statements. This method will create a prepared statement for you and return a cursor object that will get a couple rows at a time from the database. This is great for select statements with lots of results. You can also use query without a cursor and transaction by using await

```
from asyncpgsa import pg

# using a cursor and and transaction isolation
async with pg.query(select_statement) as cursor:
    async for row in cursor:
        a = row['col_name']

# no cursor or isolation, all results at once
results = await pg.query(select_statement)
for row in results:
    a = row['col_name']
```

### 4.2.3 fetch

Want to run a simple statement and get the results as a list? Fetch is for you.

```
from asyncpgsa import pg

for row in await pg.fetch(query):
    a = row['col_name']
```

### 4.2.4 fetchrow

This is just like fetch, but only returns a single row. Good for insert/update/delete calls.

```
from asyncpgsa import pg

row = await pg.fetchrow(query)
a = row['col_name']
```

### 4.2.5 fetchval

Like fetch row but also only a single column. Dont bother getting the whole row when you only need a single value

Column is a 0 index value.

```
from asyncpgsa import pg

value = await pg.fetchval(query, column=0)
```

### 4.2.6 Transaction

Everything is wrapped in a transaction for you, but if you need to do multiple things in a single transaction, then establish a transaction using an `async with` block. Commits and rollbacks will be handled for you.

```
from asyncpgsa import pg

async with pg.transaction() as conn:
    for row in await conn.fetch(query):
        a =  row['col_name']

    await conn.fetchval(update_query)
```

### 4.2.7 Begin

Begin is the same as transaction, you just get to choose which word you like best

```
from asyncpgsa import pg

async with pg.begin() as conn:
    for row in await conn.fetch(query):
        a =  row['col_name']

    await conn.fetchval(update_query)
```

## 4.3 Pool

If you dont mind passing around the pool object, you can use a pool directly. With the pool object, you currently have to wrap everything in a transaction.

### 4.3.1 Creating the pool

```
import asyncpgsa
pool = await asyncpgsa.create_pool(
    host=HOST,
    port=PORT,
    database=DATABASE,
    user=USER,
    # loop=event_loop,
    password=PASS,
    min_size=5,
    max_size=10
)
```

### 4.3.2 Transaction

The transaction context manager will establish a connection and start a transaction all at once. It returns the connection object. Commits and rollbacks will be handled for you.

```
async with pool.transaction() as conn:
    # do something with conn
    # when your code block is done, rollback/commit will happen automatically
```

### 4.3.3 fetch

Want to run a simple statement and get the results as a list? Fetch is for you.

```
#No transaction
async with pool.acquire() as conn:
    for row in await conn.fetch(query):
        a = row['col_name']

#with transaction
async with pool.transaction() as conn:
        result = await conn.fetch(query)

    for row in result:
        a = row['col_name']
```

### 4.3.4 fetchrow

This is just like fetch, but only returns a single row. Good for insert/update/delete calls.

```
async with pool.transaction() as conn:
    row = await conn.fetchrow(query)

a = row['col_name']
```

### 4.3.5 fetchval

Like fetch row but also only a single column. Dont bother getting the whole row when you only need a single value

Column is a 0 index value.

```
async with pool.transaction() as conn:
    value = await conn.fetchval(query, column=0)
```

### 4.3.6 json

```
import json
import ujson
# NOTE: ujson is very fast but ujson.dumps is not safe.

async def main():
     async def set_json_charset(connection):
         await connection.set_type_codec(
             'json',
             encoder=json.dumps,
             decoder=ujson.loads,
             schema='pg_catalog'
         )

     await pg.init("postgresql://127.0.0.1/template0", init=set_json_charset)

     ...
```

As another option you can initialize PostgreSQL dialect with custom JSON serializer and deserializer and pass it into *pg.init*

```
from asyncpgsa.connection import get_dialect

async def main():
    dialect = get_dialect(
        json_serializer=json.dumps,
        json_deserializer=ujson.loads
    )

    await pg.init("postgresql://127.0.0.1/template0", dialect=dialect)

    ...
```

Also you can initialize pool with custom dialect

```
import asyncpgsa
from asyncpgsa.connection import get_dialect

async def main():
    dialect = get_dialect(
        json_serializer=json.dumps,
        json_deserializer=ujson.loads
    )

    await asyncpgsa.create_pool(
        dialect=dialect,
        ...
    )

    ...
```

CHAPTER 5

# Compile

If you just want to roll you own everything and use asyncpg raw without all these wrappers, you can probably do it by just using the compile method in this repo

```
import asyncpgsa

query = sa.select('*').select_from(sa.text('mt_table'))
query_string, params = asyncpgsa.compile_query(query)
# Now you have the raw query string ready for asyncpg, and the ordered parameters.
results = await asyncpg_connection.fetch(query_string, params)
```

CHAPTER 6

# Testing

The library includes a testing library as well to make testing in your app easier. The testing module mocks out all the database calls and never actually hits a database.

## 6.1 Setting up

In order to setup a mock, all you need to do is use the *MockPG* module in *asyncpgsa.testing* Then you need to set the responses that you are expecting. Setting the responses is done by calling *mock_pg.set_database_results()* where every argument is a list of dictionaries.

### 6.1.1 Example test

Here is an example test.

```
from asyncpgsa.testing import MockPG

async def test_run_query(monkeypatch)
   pg = MockPG()
   pg.set_database_results([{'id': 1}, {'id': 2}])
   monkeypatch.setattr('mypackage.mymodule.pg', pg)
   results = await mymodule.run_query()

   assert results[0].id == 1
   assert results[1].id == 2
```

And another where there are multiple queries

```
from asyncpgsa.testing import MockPG

async def test_run_query(monkeypatch)
   pg = MockPG()
```

(continues on next page)

(continued from previous page)

```
    pg.set_database_results([{'id': 1}, {'id': 2}],
                            [{'id': 28, 'name': 'bob'])
    monkeypatch.setattr('mypackage.mymodule.pg', pg)
    results = await mymodule.run_multiple_queries()

    assert results[0].id == 1
    assert results[1].id == 2
    assert results[2].name == 'bob'
```