peewee-async Documentation

Release 0.6.1a

Alexey Kinev

Dec 18, 2018

Contents

peewee-async is a library providing asynchronous interface powered by asyncio for peewee ORM.

Current state: **alpha**, yet API seems fine and mostly stable.

In current version (0.5.x) new-high level API is introduced while older low-level API partially marked as deprecated.

- Works on Python 3.4+
- Has support for PostgreSQL via *aiopg*
- Has support for MySQL via *aiomysql*
- Single point for high-level async API
- Drop-in replacement for sync code, sync will remain sync
- Basic operations are supported
- Transactions support is present, yet not heavily tested

The source code is hosted on GitHub.

CHAPTER 1

Quickstart

```
import asyncio
import peewee
import peewee_async

# Nothing special, just define model and database:

database = peewee_async.PostgresqlDatabase('test')

class TestModel(peewee.Model):
    text = peewee.CharField()

    class Meta:
        database = database

# Look, sync code is working!

TestModel.create_table(True)
TestModel.create(text="Yo, I can do it sync!")
database.close()

# Create async models manager:

objects = peewee_async.Manager(database)

# No need for sync anymore!

database.set_allow_sync(False)

async def handler():
    await objects.create(TestModel, text="Not bad. Watch this, I'm async!")
    all_objects = await objects.execute(TestModel.select())
    for obj in all_objects:
        print(obj.text)
```

(continues on next page)

(continued from previous page)

```
loop = asyncio.get_event_loop()
loop.run_until_complete(handler())
loop.close()

# Clean up, can do it sync again:
with objects.allow_sync():
    TestModel.drop_table(True)

# Expected output:
# Yo, I can do it sync!
# Not bad. Watch this, I'm async!
```

CHAPTER 2

Install

Install latest version from PyPI.

For PostgreSQL:

```
pip install peewee-async aiopg
```

For MySQL:

```
pip install peewee-async aiomysql
```

2.1 Install from sources

```
git clone https://github.com/05bit/peewee-async.git
cd peewee-async
python setup.py install
```

2.2 Running tests

Prepare environment for tests:

- Clone source code from GitHub as shown above
- Create PostgreSQL database for testing, i.e. named 'test'
- Create `tests.json` config file based on `tests.json.sample`

Then run tests:

```
python setup.py test
```

CHAPTER 3

Report bugs and discuss

You are welcome to add discussion topics or bug reports to tracker on GitHub!

CHAPTER 4

Contents

4.1 High-level (new) API

High-level API provides a single point for all async ORM calls. Meet the *Manager* class! The idea of Manager originally comes from Django, but it's redesigned to meet new asyncio patterns.

First of all, once Manager is initialized with database and event loop, it's easy and safe to perform async calls. And all async operations and transactions management methods are bundled with a single object. No need to pass around database instance, event loop, etc.

Also there's no need to connect and re-connect before executing async queries with manager! It's all automatic. But you can run Manager.connect() or Manager.close() when you need it.

Note: code examples below are written for Python 3.5.x, it is possible to adapt them for Python 3.4.x by replacing *await* with *yield from* and *async def* with *@asyncio.coroutine* decorator. And async context managers like transaction() etc, are only possible in Python 3.5+

OK, let's provide an example:

```
import asyncio
import peewee
import logging
from peewee_async import Manager, PostgresqlDatabase

loop = asyncio.new_event_loop() # Note: custom loop!
database = PostgresqlDatabase('test')
objects = Manager(database, loop=loop)
```

– once objects is created with specified loop, all database connections **automatically** will be set up on **that loop**. Sometimes, it's so easy to forget to pass custom loop instance, but now it's not a problem! Just initialize with an event loop once.

Let's define a simple model:

```
class PageBlock(peewee.Model):
    key = peewee.CharField(max_length=40, unique=True)
    text = peewee.TextField(default='')

    class Meta:
        database = database
```

– as you can see, nothing special in this code, just plain `peewee.Model` definition.

Now we need to create a table for model:

```
PageBlock.create_table(True)
```

– this code is **sync**, and will do **absolutely the same thing** as would do code with regular `peewee.PostgresqlDatabase`. This is intentional, I believe there's just no need to run database initialization code asynchronously! *Less code, less errors*.

From now we may want **only async** calls and treat sync as unwanted or as errors:

```
objects.database.allow_sync = False # this will raise AssertionError on ANY sync call
```

– alternatevely we can set `ERROR` or `WARNING` loggin level to `database.allow_sync`:

```
objects.database.allow_sync = logging.ERROR
```

Finally, let's do something async:

```
async def my_async_func():
    # Add new page block
    await objects.create_or_get(
        PageBlock, key='title',
        text="Peewee is AWESOME with async!")

    # Get one by key
    title = await objects.get(PageBlock, key='title')
    print("Was:", title.text)

    # Save with new text
    title.text = "Peewee is SUPER awesome with async!"
    await objects.update(title)
    print("New:", title.text)

loop.run_until_complete(my_async_func())
loop.close()
```

That's it!

Other methods for operations like selecting, deleting etc. are listed below.

4.1.1 Manager

class peewee_async.**Manager**(*database=None*, *, *loop=None*)

Async peewee models manager.

Parameters

- **loop** – (optional) asyncio event loop
- **database** – (optional) async database driver

Example:

```
class User(peewee.Model):
    username = peewee.CharField(max_length=40, unique=True)

objects = Manager(PostgresqlDatabase('test'))

async def my_async_func():
    user0 = await objects.create(User, username='test')
    user1 = await objects.get(User, id=user0.id)
    user2 = await objects.get(User, username='test')
    # All should be the same
    print(user1.id, user2.id, user3.id)
```

If you don't pass database to constructor, you should define `database` as a class member like that:

```
database = PostgresqlDatabase('test')

class MyManager(Manager):
    database = database

objects = MyManager()
```

`Manager.`**`database = None`**
Async database driver for manager. Must be provided in constructor or as a class member.

`Manager.`**`allow_sync`**`()`
Allow sync queries within context. Close the sync database connection on exit if connected.

Example:

```
with objects.allow_sync():
    PageBlock.create_table(True)
```

`Manager.`**`get`**`(`*source_*, **args*, ***kwargs*`)`
Get the model instance.

Parameters **`source`** – model or base query for lookup

Example:

```
async def my_async_func():
    obj1 = await objects.get(MyModel, id=1)
    obj2 = await objects.get(MyModel, MyModel.id==1)
    obj3 = await objects.get(MyModel.select().where(MyModel.id==1))
```

All will return *MyModel* instance with *id = 1*

`Manager.`**`create`**`(`*model_*, ***data*`)`
Create a new object saved to database.

`Manager.`**`update`**`(`*obj*, *only=None*`)`
Update the object in the database. Optionally, update only the specified fields. For creating a new object use *`create()`*

Parameters **`only`** – (optional) the list/tuple of fields or field names to update

`Manager.`**`delete`**`(`*obj*, *recursive=False*, *delete_nullable=False*`)`
Delete object from database.

Manager.**get_or_create**(*model_*, *defaults=None*, ***kwargs*)
Try to get an object or create it with the specified defaults.

Return 2-tuple containing the model instance and a boolean indicating whether the instance was created.

Manager.**create_or_get**(*model_*, ***kwargs*)
Try to create new object with specified data. If object already exists, then try to get it by unique fields.

Manager.**execute**(*query*)
Execute query asyncronously.

Manager.**prefetch**(*query*, **subqueries*)
Asynchronous version of the *prefetch()* from peewee.

Returns Query that has already cached data for subqueries

Manager.**count**(*query*, *clear_limit=False*)
Perform *COUNT* aggregated query asynchronously.

Returns number of objects in `select()` query

Manager.**scalar**(*query*, *as_tuple=False*)
Get single value from `select()` query, i.e. for aggregation.

Returns result is the same as after sync `query.scalar()` call

Manager.**connect**()
Open database async connection if not connected.

Manager.**close**()
Close database async connection if connected.

Manager.**atomic**()
Similar to *peewee.Database.atomic()* method, but returns **asynchronous** context manager.

Example:

```
async with objects.atomic():
    await objects.create(
        PageBlock, key='intro',
        text="There are more things in heaven and earth, "
             "Horatio, than are dreamt of in your philosophy.")
    await objects.create(
        PageBlock, key='signature', text="William Shakespeare")
```

Manager.**transaction**()
Similar to *peewee.Database.transaction()* method, but returns **asynchronous** context manager.

Manager.**savepoint**(*sid=None*)
Similar to *peewee.Database.savepoint()* method, but returns **asynchronous** context manager.

4.1.2 Databases

class peewee_async.**PostgresqlDatabase**(*database*, *thread_safe=True*, *autorollback=False*, *field_types=None*, *operations=None*, *autocommit=None*, ***kwargs*)
PosgreSQL database driver providing **single drop-in sync** connection and **single async connection** interface.

Example:

```
database = PostgresqlDatabase('test')
```

See also: http://peewee.readthedocs.io/en/latest/peewee/api.html#PostgresqlDatabase

class peewee_async.**PooledPostgresqlDatabase**(*database*, *thread_safe=True*, *autorollback=False*, *field_types=None*, *operations=None*, *autocommit=None*, ***kwargs*)

PosgreSQL database driver providing **single drop-in sync** connection and **async connections pool** interface.

Parameters **max_connections** – connections pool size

Example:

```
database = PooledPostgresqlDatabase('test', max_connections=20)
```

See also: http://peewee.readthedocs.io/en/latest/peewee/api.html#PostgresqlDatabase

class peewee_asyncext.**PostgresqlExtDatabase**(**args*, ***kwargs*)

PosgreSQL database extended driver providing **single drop-in sync** connection and **single async connection** interface.

JSON fields support is always enabled, HStore supports is enabled by default, but can be disabled with `register_hstore=False` argument.

Example:

```
database = PostgresqlExtDatabase('test', register_hstore=False)
```

See also: https://peewee.readthedocs.io/en/latest/peewee/

playhouse.html#PostgresqlExtDatabase

class peewee_asyncext.**PooledPostgresqlExtDatabase**(**args*, ***kwargs*)

PosgreSQL database extended driver providing **single drop-in sync** connection and **async connections pool** interface.

JSON fields support is always enabled, HStore supports is enabled by default, but can be disabled with `register_hstore=False` argument.

Parameters **max_connections** – connections pool size

Example:

```
database = PooledPostgresqlExtDatabase('test', register_hstore=False,
                                       max_connections=20)
```

See also: https://peewee.readthedocs.io/en/latest/peewee/

playhouse.html#PostgresqlExtDatabase

class peewee_async.**MySQLDatabase**(*database*, *thread_safe=True*, *autorollback=False*, *field_types=None*, *operations=None*, *autocommit=None*, ***kwargs*)

MySQL database driver providing **single drop-in sync** connection and **single async connection** interface.

Example:

```
database = MySQLDatabase('test')
```

See also: http://peewee.readthedocs.io/en/latest/peewee/api.html#MySQLDatabase

class peewee_async.**PooledMySQLDatabase**(*database*, *thread_safe=True*, *autorollback=False*, *field_types=None*, *operations=None*, *autocommit=None*, ***kwargs*)

MySQL database driver providing **single drop-in sync** connection and **async connections pool** interface.

Parameters **max_connections** – connections pool size

Example:

```
database = MySQLDatabase('test', max_connections=10)
```

See also: http://peewee.readthedocs.io/en/latest/peewee/api.html#MySQLDatabase

4.2 Low-level (older) API

Note: all query methods are **coroutines**.

4.2.1 Select, update, delete

peewee_async.**execute**(*query*)
Execute *SELECT*, *INSERT*, *UPDATE* or *DELETE* query asyncronously.

Parameters **query** – peewee query instance created with `Model.select()`, `Model.update()` etc.

Returns result depends on query type, it's the same as for sync `query.execute()`

peewee_async.**get_object**(*source*, **args*)
Get object asynchronously.

Parameters

- **source** – mode class or query to get object from
- **args** – lookup parameters

Returns model instance or raises `peewee.DoesNotExist` if object not found

peewee_async.**create_object**(*model*, ***data*)
Create object asynchronously.

Parameters

- **model** – mode class
- **data** – data for initializing object

Returns new object saved to database

peewee_async.**delete_object**(*obj*, *recursive=False*, *delete_nullable=False*)
Delete object asynchronously.

Parameters

- **obj** – object to delete
- **recursive** – if `True` also delete all other objects depends on object
- **delete_nullable** – if *True* and delete is recursive then delete even 'nullable' dependencies

For details please check out Model.delete_instance() in peewee docs.

peewee_async.**update_object**(*obj*, *only=None*)
Update object asynchronously.

Parameters

- **obj** – object to update
- **only** – list or tuple of fields to updata, is *None* then all fields updated

This function does the same as Model.save() for already saved object, but it doesn't invoke `save()` method on model class. That is important to know if you overrided save method for your model.

`peewee_async.`**`prefetch`**(*sq*, **subqueries*)
Asynchronous version of the *prefetch()* from peewee.

4.2.2 Transactions

Transactions required Python 3.5+ to work, because their syntax is based on async context managers.

Important note transactions rely on data isolation on *asyncio* per-task basis. That means, all queries for single transaction should be performed **within same task**.

`peewee_async.`**`atomic`**(*db*)
Asynchronous context manager (*async with*), similar to *peewee.atomic()*.

`peewee_async.`**`savepoint`**(*db*, *sid=None*)
Asynchronous context manager (*async with*), similar to *peewee.savepoint()*.

`peewee_async.`**`transaction`**(*db*)
Asynchronous context manager (*async with*), similar to *peewee.transaction()*. Will start new *asyncio* task for transaction if not started already.

4.2.3 Aggregation

`peewee_async.`**`count`**(*query*, *clear_limit=False*)
Perform *COUNT* aggregated query asynchronously.

Returns number of objects in `select()` query

`peewee_async.`**`scalar`**(*query*, *as_tuple=False*)
Get single value from `select()` query, i.e. for aggregation.

Returns result is the same as after sync `query.scalar()` call

4.2.4 Databases

`class` `peewee_async.`**`PostgresqlDatabase`**(*database*, *thread_safe=True*, *autorollback=False*, *field_types=None*, *operations=None*, *autocommit=None*, ***kwargs*)
PosgreSQL database driver providing **single drop-in sync** connection and **single async connection** interface.

Example:

```
database = PostgresqlDatabase('test')
```

See also: http://peewee.readthedocs.io/en/latest/peewee/api.html#PostgresqlDatabase

`atomic_async`()
Similar to peewee *Database.atomic()* method, but returns asynchronous context manager.

`connect_async`(*loop=None*, *timeout=None*)
Set up async connection on specified event loop or on default event loop.

savepoint_async(*sid=None*)
Similar to peewee *Database.savepoint()* method, but returns asynchronous context manager.

transaction_async()
Similar to peewee *Database.transaction()* method, but returns asynchronous context manager.

class peewee_async.**PooledPostgresqlDatabase**(*database*, *thread_safe=True*, *autorollback=False*, *field_types=None*, *operations=None*, *autocommit=None*, ***kwargs*)
PosgreSQL database driver providing **single drop-in sync** connection and **async connections pool** interface.

Parameters **max_connections** – connections pool size

Example:

```
database = PooledPostgresqlDatabase('test', max_connections=20)
```

See also: http://peewee.readthedocs.io/en/latest/peewee/api.html#PostgresqlDatabase

connect_async(*loop=None*, *timeout=None*)
Set up async connection on specified event loop or on default event loop.

class peewee_asyncext.**PostgresqlExtDatabase**(**args*, ***kwargs*)
PosgreSQL database extended driver providing **single drop-in sync** connection and **single async connection** interface.

JSON fields support is always enabled, HStore supports is enabled by default, but can be disabled with `register_hstore=False` argument.

Example:

```
database = PostgresqlExtDatabase('test', register_hstore=False)
```

See also: https://peewee.readthedocs.io/en/latest/peewee/

playhouse.html#PostgresqlExtDatabase

atomic_async()
Similar to peewee *Database.atomic()* method, but returns asynchronous context manager.

connect_async(*loop=None*, *timeout=None*)
Set up async connection on specified event loop or on default event loop.

savepoint_async(*sid=None*)
Similar to peewee *Database.savepoint()* method, but returns asynchronous context manager.

transaction_async()
Similar to peewee *Database.transaction()* method, but returns asynchronous context manager.

class peewee_asyncext.**PooledPostgresqlExtDatabase**(**args*, ***kwargs*)
PosgreSQL database extended driver providing **single drop-in sync** connection and **async connections pool** interface.

JSON fields support is always enabled, HStore supports is enabled by default, but can be disabled with `register_hstore=False` argument.

Parameters **max_connections** – connections pool size

Example:

```
database = PooledPostgresqlExtDatabase('test', register_hstore=False,
                                       max_connections=20)
```

See also: https://peewee.readthedocs.io/en/latest/peewee/

playhouse.html#PostgresqlExtDatabase

connect_async(*loop=None*, *timeout=None*)
Set up async connection on specified event loop or on default event loop.

4.3 Using async peewee with Tornado

Tornado is a mature and powerful asynchronous web framework. It provides its own event loop, but there's an option to run Tornado on asyncio event loop. And that's exactly what we need!

The complete working example is provided below. And here are some general notes:

1. **Be aware of current asyncio event loop!**

   In the provided example we use the default event loop everywhere, and that's OK. But if you see your application got silently stuck, that's most probably that some task is started on the different loop and will never complete as long as that loop is not running.

2. Tornado request handlers **does not** start asyncio tasks by default.

   The `CreateHandler` demostrates that, `current_task()` returns `None` until task is run explicitly.

3. Transactions **must** run within task context.

   All transaction operations have to be done within task. So if you need to run a transaction from Tornado handler, you have to wrap your call into task with `create_task()` or `ensure_future()`.

   Also note: if you spawn an extra task during a transaction, it will run outside of that transaction.

```
import tornado.web
import logging
import peewee
import asyncio
import peewee_async

# Set up database and manager
database = peewee_async.PooledPostgresqlDatabase('test')

# Define model
class TestNameModel(peewee.Model):
    name = peewee.CharField()
    class Meta:
        database = database

    def __str__(self):
        return self.name

# Create table, add some instances
TestNameModel.create_table(True)
TestNameModel.get_or_create(id=1, defaults={'name': "TestNameModel id=1"})
TestNameModel.get_or_create(id=2, defaults={'name': "TestNameModel id=2"})
TestNameModel.get_or_create(id=3, defaults={'name': "TestNameModel id=3"})
database.close()

# Set up Tornado application on asyncio
from tornado.platform.asyncio import AsyncIOMainLoop
AsyncIOMainLoop().install()
app = tornado.web.Application(debug=True)
app.listen(port=8888)
```

(continues on next page)

(continued from previous page)

```
app.objects = peewee_async.Manager(database)

# Add handlers
class RootHandler(tornado.web.RequestHandler):
    """Accepts GET and POST methods.

    POST: create new instance, `name` argument is required
    GET: get instance by id, `id` argument is required
    """
    async def post(self):
        name = self.get_argument('name')
        obj = await self.application.objects.create(TestNameModel, name=name)
        self.write({
            'id': obj.id,
            'name': obj.name
        })

    async def get(self):
        obj_id = self.get_argument('id', None)

        if not obj_id:
            self.write("Please provide the 'id' query argument, i.e. ?id=1")
            return

        try:
            obj = await self.application.objects.get(TestNameModel, id=obj_id)
            self.write({
                'id': obj.id,
                'name': obj.name,
            })
        except TestNameModel.DoesNotExist:
            raise tornado.web.HTTPError(404, "Object not found!")

class CreateHandler(tornado.web.RequestHandler):
    async def get(self):
        loop = asyncio.get_event_loop()
        task1 = asyncio.Task.current_task() # Just to demonstrate it's None
        task2 = loop.create_task(self.get_or_create())
        obj = await task2
        self.write({
            'task1': task1 and id(task1),
            'task2': task2 and id(task2),
            'obj': str(obj),
            'text': "'task1' should be null, "
                    "'task2' should be not null, "
                    "'obj' should be newly created object",
        })

    async def get_or_create(self):
        obj_id = self.get_argument('id', None)
        async with self.application.objects.atomic():
            obj, created = await self.application.objects.get_or_create(
                TestNameModel, id=obj_id,
                defaults={'name': "TestNameModel id=%s" % obj_id})
            return obj

app.add_handlers('', [
```

(continues on next page)

(continued from previous page)

```
    (r"/", RootHandler),
    (r"/create", CreateHandler),
])

# Setup verbose logging
log = logging.getLogger('')
log.addHandler(logging.StreamHandler())
log.setLevel(logging.DEBUG)

# Run loop
print("""Run application server http://127.0.0.1:8888

    Try GET urls:
    http://127.0.0.1:8888?id=1
    http://127.0.0.1:8888/create?id=100

    Try POST with name=<some text> data:
    http://127.0.0.1:8888

^C to stop server""")
loop = asyncio.get_event_loop()
try:
    loop.run_forever()
except KeyboardInterrupt:
    print(" server stopped")
```

4.4 More examples

TODO: update examples to high-level API.

4.4.1 Using both sync and async calls

```
import asyncio
import peewee
import peewee_async

database = peewee_async.PostgresqlDatabase('test')
loop = asyncio.get_event_loop()

class TestModel(peewee.Model):
    text = peewee.CharField()

    class Meta:
        database = database

# Create table synchronously!
TestModel.create_table(True)
# This is optional: close sync connection
database.close()

@asyncio.coroutine
def my_handler():
```

(continues on next page)

(continued from previous page)

```
    obj1 = TestModel.create(text="Yo, I can do it sync!")
    obj2 = yield from peewee_async.create_object(TestModel, text="Not bad. Watch this,
↪ I'm async!")

    all_objects = yield from peewee_async.execute(TestModel.select())
    for obj in all_objects:
        print(obj.text)

    obj1.delete_instance()
    yield from peewee_async.delete_object(obj2)

loop.run_until_complete(database.connect_async(loop=loop))
loop.run_until_complete(my_handler())
```

4.4.2 Using transactions

```
import asyncio
import peewee
import peewee_async

# ... some init code ...

async def test():
    obj = await create_object(TestModel, text='FOO')
    obj_id = obj.id

    try:
        async with database.atomic_async():
            obj.text = 'BAR'
            await update_object(obj)
            raise Exception('Fake error')
    except:
        res = await get_object(TestModel, TestModel.id == obj_id)

    print(res.text) # Should print 'FOO', not 'BAR'

loop.run_until_complete(test())
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index