Piccolo

Release 0.21.2

Jun 24, 2021

Contents:

CHAPTER 1

Getting Started

1.1 What is Piccolo?

Piccolo is a fast, easy to learn ORM and query builder.

Some of it's stand out features are:

- Support for sync and async - see *Sync and Async*.
- A builtin playground, which makes learning a breeze - see *Playground*.
- Works great with iPython and VSCode - see *Tab Completion*.
- Batteries included - a *User model and authentication*, *migrations*, an *admin*, and more.
- Templates for creating your own *ASGI web app*.

1.2 Database Support

Postgres is the primary database which Piccolo was designed for.

Limited SQLite support is available, mostly to enable tooling like the *playground*. Postgres is the only database we recommend for use in production with Piccolo.

1.3 Installing Piccolo

1.3.1 Python

You need Python 3.7 or above installed on your system.

1.3.2 Pip

Now install piccolo, ideally inside a virtualenv:

```
# Optional - creating a virtualenv on Unix:
python3.7 -m venv my_project
cd my_project
source bin/activate

# The important bit:
pip install piccolo

# For optional orjson support, which improves JSON serialisation
# performance:
pip install piccolo[orjson]
```

1.4 Playground

Piccolo ships with a handy command called *playground*, which is a great way to learn the basics.

```
piccolo playground run
```

It will create an example schema for you (see *Example Schema*) , populates it with data, and launches an iPython shell.

You can follow along with the tutorials without first learning advanced concepts like migrations.

It's a nice place to experiment with querying / inserting / deleting data using Piccolo, no matter how experienced you are.

Warning: Each time you launch the playground it flushes out the existing tables and rebuilds them, so don't use it for anything permanent!

1.4.1 SQLite

SQLite is used by default, which provides a zero config way of getting started.

A `piccolo.sqlite` file will get created in the current directory.

1.4.2 Advanced usage

To see how to use the playground with Postgres, and other advanced usage, see *Advanced Playground Usage*.

1.4.3 Test queries

The schema generated in the playground represents fictional bands and their concerts.

When the playground is started it prints out the available tables.

Give these queries a go:

```
Band.select().run_sync()
Band.objects().run_sync()
Band.select(Band.name).run_sync()
Band.select(Band.name, Band.manager.name).run_sync()
```

1.4.4 Tab completion is your friend

Piccolo was designed to make tab completion available in as many situations as possible. Use it to find the column names for a table (e.g. `Band.name`), and the different query types (e.g. `Band.select`).

Using tab completion will help avoid errors, and speed up your coding.

1.5 Setup Postgres

1.5.1 Installation

Mac

The quickest way to get Postgres up and running on the Mac is using Postgres.app.

Ubuntu

On Ubuntu you can use `apt`.

```
sudo apt update
sudo apt install postgresql
```

1.5.2 Creating a database

Mac

psql

Postgres.app should make `psql` available for the user who installed it.

```
psql
```

Enter the following:

```
CREATE DATABASE "my_database_name";
```

pgAdmin

If you prefer a GUI, pgAdmin has an installer available.

Ubuntu

psql

Using `psql`:

```
sudo su postgres -c psql
```

Enter the following:

```
CREATE DATABASE "my_database_name";
```

pgAdmin

DEB packages are available for Ubuntu.

1.5.3 Postgres version

Piccolo is currently tested against Postgres 9.6, 10.6, and 11.1 so it's recommended to use one of those. To check all supported versions, see the Travis file.

1.5.4 What about other databases?

At the moment the focus is on providing the best Postgres experience possible, along with some SQLite support. Other databases may be supported in the future.

1.6 Sync and Async

One of the main motivations for making Piccolo was the lack of options for ORMs which support asyncio.

However, you can use Piccolo in synchronous apps as well, whether that be a WSGI web app, or a data science script.

1.6.1 Sync example

```
from my_schema import Band


def main():
    print(Band.select().run_sync())


if __name__ == '__main__':
    main()
```

1.6.2 Async example

```
import asyncio
from my_schema import Band


async def main():
    print(await Band.select().run())


if __name__ == '__main__':
    asyncio.run(main())
```

1.6.3 Which to use?

A lot of the time, using the sync version works perfectly fine. Many of the examples use the sync version.

Using the async version is useful for web applications which require high throughput, based on ASGI frameworks. Piccolo makes building an ASGI web app really simple - see *ASGI*.

1.6.4 Explicit

By using `run` and `run_sync`, it makes it very explicit when a query is actually being executed.

Until you execute one of those methods, you can chain as many methods onto your query as you like, safe in the knowledge that no database queries are being made.

1.7 Example Schema

This is the schema used by the example queries throughout the docs.

```
from piccolo.table import Table
from piccolo.columns import ForeignKey, Integer, Varchar


class Manager(Table):
    name = Varchar(length=100)


class Band(Table):
    name = Varchar(length=100)
    manager = ForeignKey(references=Manager)
    popularity = Integer()
```

To understand more about defining your own schemas, see *Defining a Schema*.

CHAPTER 2

Query Types

There are many different queries you can perform using Piccolo.

The main ways to query data are with *Select*, which returns data as dictionaries, and *Objects* , which returns data as class instances, like a typical ORM.

2.1 Select

Hint: Follow along by installing Piccolo and running *piccolo playground run* - see *Playground*

To get all rows:

```
>>> Band.select().run_sync()
[{'id': 1, 'name': 'Pythonistas', 'manager': 1, 'popularity': 1000},
 {'id': 2, 'name': 'Rustaceans', 'manager': 2, 'popularity': 500}]
```

To get certain columns:

```
>>> Band.select(Band.name).run_sync()
[{'name': 'Rustaceans'}, {'name': 'Pythonistas'}]
```

Or making an alias to make it shorter:

```
>>> b = Band
>>> b.select(b.name).run_sync()
[{'id': 1, 'name': 'Pythonistas', 'manager': 1, 'popularity': 1000},
 {'id': 2, 'name': 'Rustaceans', 'manager': 2, 'popularity': 500}]
```

Hint: All of these examples also work with async by using .run() inside coroutines - see *Sync and Async*.

2.1.1 as_alias

By using `as_alias`, the name of the row can be overriden in the response.

```
>>> Band.select(Band.name.as_alias('title')).run_sync()
[{'title': 'Rustaceans'}, {'title': 'Pythonistas'}]
```

This is equivalent to `SELECT name AS title FROM band` in SQL.

2.1.2 Joins

One of the most powerful things about select is it's support for joins.

```
>>> b = Band
>>> b.select(b.name, b.manager.name).run_sync()
[{'name': 'Pythonistas', 'manager.name': 'Guido'}, {'name': 'Rustaceans', 'manager.
↪name': 'Graydon'}]
```

The joins can go several layers deep.

```
c = Concert
c.select(
    c.id,
    c.band_1.manager.name
).run_sync()
```

2.1.3 String syntax

Alternatively, you can specify the column names using a string. The disadvantage is you won't have tab completion, but sometimes it's more convenient.

```
Band.select('name').run_sync()

# For joins:
Band.select('manager.name').run_sync()
```

2.1.4 Query clauses

batch

See *batch*.

columns

By default all columns are returned from the queried table.

```
b = Band
# Equivalent to SELECT * from band
b.select().run_sync()
```

To restrict the returned columns, either pass in the columns into the `select` method, or use the *columns* method.

```
b = Band
# Equivalent to SELECT name from band
b.select().columns(b.name).run_sync()
```

The *columns* method is additive, meaning you can chain it to add additional columns.

```
b = Band
b.select().columns(b.name).columns(b.manager).run_sync()

# Or just define it one go:
b.select().columns(b.name, b.manager).run_sync()
```

first

See *first*.

group_by

See *group_by*.

limit

See *limit*.

offset

See *offset*.

order_by

See *order_by*.

output

By default, the data is returned as a list of dictionaries (where each dictionary represents a row). This can be altered using the `output` method.

To return the data as a JSON string:

```
>>> b = Band
>>> b.select().output(as_json=True).run_sync()
'[{"name":"Pythonistas","manager":1,"popularity":1000,"id":1},{"name":"Rustaceans",
↪"manager":2,"popularity":500,"id":2}]'
```

Piccolo can use orjson for JSON serialisation, which is blazing fast, and can handle most Python types, including dates, datetimes, and UUIDs. To install Piccolo with orjson support use `pip install piccolo[orjson]`.

where

See *where*.

2.2 Objects

When doing *Select* queries, you get data back in the form of a list of dictionaries (where each dictionary represents a row). This is useful in a lot of situations, but it's sometimes preferable to get objects back instead, as we can manipulate them, and save the changes back to the database.

In Piccolo, an instance of a `Table` class represents a row. Let's do some examples.

2.2.1 Fetching objects

To get all objects:

```
>>> Band.objects().run_sync()
[<Band: 1>, <Band: 2>]
```

To get certain rows:

```
>>> Band.objects().where(Band.name == 'Pythonistas').run_sync()
[<Band: 1>]
```

To get the first row:

```
>>> Band.objects().first().run_sync()
<Band: 1>
```

You'll notice that the API is similar to *Select* - except it returns all columns.

2.2.2 Creating objects

```
>>> band = Band(name="C-Sharps", popularity=100)
>>> band.save().run_sync()
```

2.2.3 Updating objects

Objects have a `save` method, which is convenient for updating values:

```
pythonistas = Band.objects().where(
    Band.name == 'Pythonistas'
).first().run_sync()

pythonistas.popularity = 100000
pythonistas.save().run_sync()
```

2.2.4 Deleting objects

Similarly, we can delete objects, using the `remove` method.

```
pythonistas = Band.objects().where(
    Band.name == 'Pythonistas'
).first().run_sync()

pythonistas.remove().run_sync()
```

2.2.5 get_related

If you have an object with a foreign key, and you want to fetch the related object, you can do so using `get_related`.

```
pythonistas = Band.objects().where(
    Band.name == 'Pythonistas'
).first().run_sync()

manager = pythonistas.get_related(Band.manager).run_sync()
>>> print(manager.name)
'Guido'
```

2.2.6 Query clauses

batch

See *batch*.

limit

See *limit*.

offset

See *offset*.

first

See *first*.

order_by

See *order_by*.

where

See *where* .

2.3 Alter

This is used to modify an existing table.

Hint: You can use migrations instead of manually altering the schema - see *Migrations*.

2.3.1 add_column

Used to add a column to an existing table.

```
Band.alter().add_column('members', Integer()).run_sync()
```

2.3.2 drop_column

Used to drop an existing column.

```
Band.alter().drop_column('popularity').run_sync()
```

2.3.3 drop_table

Used to drop the table - use with caution!

```
Band.alter().drop_table().run_sync()
```

2.3.4 rename_column

Used to rename an existing column.

```
Band.alter().rename_column(Band.popularity, 'rating').run_sync()
```

2.3.5 set_null

Set whether a column is nullable or not.

```
# To make a row nullable:
Band.alter().set_null(Band.name, True).run_sync()

# To stop a row being nullable:
Band.alter().set_null(Band.name, False).run_sync()
```

2.3.6 set_unique

Used to change whether a column is unique or not.

```
# To make a row unique:
Band.alter().set_unique(Band.name, True).run_sync()

# To stop a row being unique:
Band.alter().set_unique(Band.name, False).run_sync()
```

2.4 Count

Returns the number of rows which match the query.

```
>>> Band.count().where(Band.name == 'Pythonistas').run_sync()
1
```

2.4.1 Query clauses

where

See *where*.

2.5 Create Table

This creates the table and columns in the database.

Hint: You can use migrations instead of manually altering the schema - see *Migrations*.

```
>>> Band.create_table().run_sync()
[]
```

To prevent an error from being raised if the table already exists:

```
>>> Band.create_table(if_not_exists=True).run_sync()
[]
```

2.6 Delete

This deletes any matching rows from the table.

```
>>> Band.delete().where(Band.name == 'Rustaceans').run_sync()
[]
```

2.6.1 force

Piccolo won't let you run a delete query without a where clause, unless you explicitly tell it to do so. This is to help prevent accidentally deleting all the data from a table.

```
>>> Band.delete().run_sync()
Raises: DeletionError

# Works fine:
>>> Band.delete(force=True).run_sync()
[]
```

2.6.2 Query clauses

where

See *where*

2.7 Exists

This checks whether any rows exist which match the criteria.

```
>>> Band.exists().where(Band.name == 'Pythonistas').run_sync()
True
```

2.7.1 Query clauses

where

See *where*.

2.8 Insert

This is used to insert rows into the table.

```
>>> Band.insert(Band(name="Pythonistas")).run_sync()
[{'id': 3}]
```

We can insert multiple rows in one go:

```
Band.insert(
    Band(name="Darts"),
    Band(name="Gophers")
).run_sync()
```

2.8.1 add

You can also compose it as follows:

```
Band.insert().add(
    Band(name="Darts")
).add(
    Band(name="Gophers")
).run_sync()
```

2.9 Raw

Should you need to, you can execute raw SQL.

```
>>> Band.raw('select * from band').run_sync()
[{'name': 'Pythonistas', 'manager': 1, 'popularity': 1000, 'id': 1},
    {'name': 'Rustaceans', 'manager': 2, 'popularity': 500, 'id': 2}]
```

It's recommended that you parameterise any values. Use curly braces `{}` as placeholders:

```
>>> Band.raw('select * from band where name = {}', 'Pythonistas').run_sync()
[{'name': 'Pythonistas', 'manager': 1, 'popularity': 1000, 'id': 1}]
```

Warning: Be careful to avoid SQL injection attacks. Don't add any user submitted data into your SQL strings, unless it's parameterised.

2.10 Update

This is used to update any rows in the table which match the criteria.

```
>>> Band.update({
>>>     Band.name: 'Pythonistas 2'
>>> }).where(
>>>     Band.name == 'Pythonistas'
>>> ).run_sync()
[]
```

As well as replacing values with new ones, you can also modify existing values, for instance by adding to an integer.

2.10.1 Modifying values

Integer columns

You can add / subtract / multiply / divide values:

```
# Add 100 to the popularity of each band:
Band.update({
    Band.popularity: Band.popularity + 100
}).run_sync()

# Decrease the popularity of each band by 100.
Band.update({
    Band.popularity: Band.popularity - 100
}).run_sync()

# Multiply the popularity of each band by 10.
Band.update({
    Band.popularity: Band.popularity * 10
}).run_sync()

# Divide the popularity of each band by 10.
Band.update({
    Band.popularity: Band.popularity / 10
}).run_sync()

# You can also use the operators in reverse:
Band.update({
    Band.popularity: 2000 - Band.popularity
}).run_sync()
```

Varchar / Text columns

You can concatenate values:

```
# Append "!!!" to each band name.
Band.update({
    Band.name: Band.name + "!!!"
}).run_sync()

# Concatenate the values in each column:
Band.update({
    Band.name: Band.name + Band.name
}).run_sync()

# Prepend "!!!" to each band name.
Band.update({
    Band.popularity: "!!!" + Band.popularity
}).run_sync()
```

You can currently only combine two values together at a time.

2.10.2 Query clauses

where

See *where*.

2.11 Transactions

Transactions allow multiple queries to be committed only once successful.

This is useful for things like migrations, where you can't have it fail in an inbetween state.

2.11.1 Atomic

This is useful when you want to programmatically add some queries to the transaction before running it.

```
transaction = Band._meta.db.atomic()
transaction.add(Manager.create_table())
transaction.add(Concert.create_table())
await transaction.run()

# You're also able to run this synchronously:
transaction.run_sync()
```

2.11.2 Transaction

This is the preferred way to run transactions - it currently only works with async.

```
async with Band._meta.db.transaction():
    await Manager.create_table().run()
    await Concert.create_table().run()
```

If an exception is raised within the body of the context manager, then the transaction is automatically rolled back. The exception is still propagated though.

2.12 Comparisons

If you're familiar with other ORMs, here are some guides which show the Piccolo equivalents of common queries.

2.12.1 Django Comparison

Here are some common queries, showing how they're done in Django vs Piccolo. All of the Piccolo examples can also be run *asynchronously*.

Queries

create

```
# Django
>>> band = Band(name="Pythonistas")
>>> band.save()
>>> band
<Band: 1>

# Piccolo
>>> band = Band(name="Pythonistas")
>>> band.save().run_sync()
>>> band
<Band: 1>
```

update

```
# Django
>>> band = Band.objects.get(name="Pythonistas")
>>> band
<Band: 1>
>>> band.name = "Amazing Band"
>>> band.save()

# Piccolo
>>> band = Band.objects().where(Band.name == 'Pythonistas').first().run_sync()
>>> band
<Band: 1>
>>> band.name = "Amazing Band"
>>> band.save().run_sync()
```

delete

Individual rows:

```
# Django
>>> band = Band.objects.get(name="Pythonistas")
>>> band.delete()

# Piccolo
>>> band = Band.objects().where(Band.name == 'Pythonistas').first().run_sync()
>>> band.remove().run_sync()
```

In bulk:

```
# Django
>>> Band.objects.filter(popularity__lt=1000).delete()

# Piccolo
>>> Band.delete().where(Band.popularity < 1000).delete().run_sync()
```

filter

```
# Django
>>> Band.objects.filter(name="Pythonistas")
[<Band: 1>]

# Piccolo
>>> Band.objects().where(Band.name == "Pythonistas").run_sync()
[<Band: 1>]
```

values_list

```
# Django
>>> Band.objects.values_list('name')
[{'name': 'Pythonistas'}, {'name': 'Rustaceans'}]

# Piccolo
>>> Band.select(Band.name).run_sync()
[{'name': 'Pythonistas'}, {'name': 'Rustaceans'}]
```

With `flat=True`:

```
# Django
>>> Band.objects.values_list('name', flat=True)
['Pythonistas', 'Rustaceans']

# Piccolo
>>> Band.select(Band.name).output(as_list=True).run_sync()
['Pythonistas', 'Rustaceans']
```

Database Settings

In Django you configure your database in `settings.py`. With Piccolo, you define an `Engine` in `piccolo_conf.py`. See *Engines*.

CHAPTER 3

Query Clauses

Query clauses are used to modify a query by making it more specific, or by modifying the return values.

3.1 first

You can use `first` clauses with the following queries:

- *Objects*
- *Select*

Rather than returning a list of results, just the first result is returned.

```
>>> Band.select().first().run_sync()
{'name': 'Pythonistas', 'manager': 1, 'popularity': 1000, 'id': 1}
```

Likewise, with objects:

```
>>> Band.objects().first().run_sync()
<Band at 0x10fdef1d0>
```

If no match is found, then *None* is returned instead.

3.2 group_by

You can use `group_by` clauses with the following queries:

- *Select*

It is used in combination with aggregate functions - `Count` is currently supported.

3.2.1 Count

In the following query, we get a count of the number of bands per manager:

```
>>> from piccolo.query.methods.select import Count

>>> b = Band
>>> b.select(
>>>     b.manager.name,
>>>     Count(b.manager)
>>> ).group_by(
>>>     b.manager
>>> ).run_sync()

[
    {"manager.name": "Graydon", "count": 1},
    {"manager.name": "Guido", "count": 1}
]
```

class piccolo.query.methods.select.**Count**(*column: Optional[piccolo.columns.base.Column] = None*)

Used in conjunction with the `group_by` clause in `Select` queries.

If a column is specified, the count is for non-null values in that column. If no column is specified, the count is for all rows, whether they have null values or not.

3.3 limit

You can use `limit` clauses with the following queries:

- *Objects*
- *Select*

Rather than returning all of the matching results, it will only return the number you ask for.

```
Band.select().limit(2).run_sync()
```

Likewise, with objects:

```
Band.objects().limit(2).run_sync()
```

3.4 offset

You can use `offset` clauses with the following queries:

- *Objects*
- *Select*

This will omit the first X rows from the response.

It's highly recommended to use it along with an *order_by* clause, otherwise the results returned could be different each time.

```
>>> Band.select(Band.name).offset(1).order_by(Band.name).run_sync()
[{'name': 'Pythonistas'}, {'name': 'Rustaceans'}]
```

Likewise, with objects:

```
>>> Band.objects().offset(1).order_by(Band.name).run_sync()
[Band2, Band3]
```

3.5 order_by

You can use `order_by` clauses with the following queries:

- *Select*
- *Objects*

To order the results by a certain column (ascending):

```
b = Band
b.select().order_by(
    b.name
).run_sync()
```

To order by descending:

```
b = Band
b.select().order_by(
    b.name,
    ascending=False
).run_sync()
```

You can order by multiple columns, and even use joins:

```
b = Band
b.select().order_by(
    b.name,
    b.manager.name
).run_sync()
```

3.6 where

You can use `where` clauses with the following queries:

- *Delete*
- *Exists*
- *Objects*
- *Select*
- *Update*

It allows powerful filtering of your data.

3.6.1 Equal / Not Equal

```
b = Band
b.select().where(
    b.name == 'Pythonistas'
).run_sync()
```

```
b = Band
b.select().where(
    b.name != 'Rustaceans'
).run_sync()
```

3.6.2 Greater than / less than

You can use the `<`, `>`, `<=`, `>=` operators, which work as you expect.

```
b = Band
b.select().where(
    b.popularity >= 100
).run_sync()
```

3.6.3 like / ilike

The percentage operator is required to designate where the match should occur.

```
b = Band
b.select().where(
    b.name.like('Py%')  # Matches the start of the string
).run_sync()

b.select().where(
    b.name.like('%istas')  # Matches the end of the string
).run_sync()

b.select().where(
    b.name.like('%is%')  # Matches anywhere in string
).run_sync()
```

`ilike` is identical, except it's case insensitive.

3.6.4 not_like

Usage is the same as `like` excepts it excludes matching rows.

```
b = Band
b.select().where(
    b.name.not_like('Py%')
).run_sync()
```

3.6.5 is_in / not_in

```
b = Band
b.select().where(
    b.name.is_in(['Pythonistas'])
).run_sync()
```

```
b = Band
b.select().where(
    b.name.not_in(['Rustaceans'])
).run_sync()
```

3.6.6 is_null / is_not_null

These queries work, but some linters will complain about doing a comparison with None:

```
b = Band

# Fetch all bands with a manager
b.select().where(
    b.manager != None
).run_sync()

# Fetch all bands without a manager
b.select().where(
    b.manager == None
).run_sync()
```

To avoid the linter errors, you can use *is_null* and *is_not_null* instead.

```
b = Band

# Fetch all bands with a manager
b.select().where(
    b.manager.is_not_null()
).run_sync()

# Fetch all bands without a manager
b.select().where(
    b.manager.is_null()
).run_sync()
```

3.6.7 Complex queries - and / or

You can make complex `where` queries using `&` for AND, and `|` for OR.

```
b = Band
b.select().where(
    (b.popularity >= 100) & (b.popularity < 1000)
).run_sync()
```

(continues on next page)

(continued from previous page)

```
b.select().where(
    (b.popularity >= 100) | (b.name ==  'Pythonistas')
).run_sync()
```

You can make really complex `where` clauses if you so choose - just be careful to include brackets in the correct place.

```
((b.popularity >= 100) & (b.manager.name ==  'Guido')) | (b.popularity > 1000)
```

Using multiple `where` clauses is equivalent to an AND.

```
b = Band

# These are equivalent:
b.select().where(
    (b.popularity >= 100) & (b.popularity < 1000)
).run_sync()

b.select().where(
    b.popularity >= 100
).where(
    b.popularity < 1000
).run_sync()
```

Using And / Or directly

Rather than using the `|` and `&` characters, you can use the `And` and `Or` classes, which are what's used under the hood.

```
from piccolo.columns.combination import And, Or

b = Band

b.select().where(
    Or(
        And(b.popularity >= 100, b.popularity < 1000),
        b.name == 'Pythonistas'
    )
).run_sync()
```

3.6.8 WhereRaw

In certain situations you may want to have raw SQL in your where clause.

```
from piccolo.columns.combination import WhereRaw

Band.select().where(
    WhereRaw("name = 'Pythonistas'")
).run_sync()
```

It's important to parameterise your SQL statements if the values come from an untrusted source, otherwise it could lead to a SQL injection attack.

```
from piccolo.columns.combination import WhereRaw

value = "Could be dangerous"

Band.select().where(
    WhereRaw("name = {}", value)
).run_sync()
```

`WhereRaw` can be combined into complex queries, just as you'd expect:

```
from piccolo.columns.combination import WhereRaw

b = Band
b.select().where(
    WhereRaw("name = 'Pythonistas'") | (b.popularity > 1000)
).run_sync()
```

3.7 batch

You can use `batch` clauses with the following queries:

- *Objects*
- *Select*

By default, a query will return as many rows as you ask it for. The problem is when you have a table containing millions of rows - you might not want to load them all into memory at once. To get around this, you can batch the responses.

```
# Returns 100 rows at a time:
async with await Manager.select().batch(batch_size=100) as batch:
    async for _batch in batch:
        print(_batch)
```

Note: `batch` is one of the few query clauses which doesn't require .run() to be used after it in order to execute. `batch` effectively replaces `run`.

There's currently no synchronous version. However, it's easy enough to achieve:

```
async def get_batch():
    async with await Manager.select().batch(batch_size=100) as batch:
        async for _batch in batch:
            print(_batch)

from piccolo.utils.sync import run_sync
run_sync(get_batch())
```

3.8 freeze

You can use the `freeze` clause with any query type.

Query.**freeze**() → piccolo.query.base.FrozenQuery

This is a performance optimisation when the same query is run repeatedly. For example:

```
TOP_BANDS = Band.select(
    Band.name
).order_by(
    Band.popularity,
    ascending=False
).limit(
    10
).output(
    as_json=True
).freeze()

# In the corresponding view/endpoint of whichever web framework
# you're using:
async def top_bands(self, request):
    return await TOP_BANDS.run()
```

It means that Piccolo doesn't have to work as hard each time the query is run to generate the corresponding SQL - some of it is cached. If the query is defined within the view/endpoint, it has to generate the SQL from scratch each time.

Once a query is frozen, you can't apply any more clauses to it (`where`, `limit`, `output` etc).

Even though `freeze` helps with performance, there are limits to how much it can help, as most of the time is still spent waiting for a response from the database. However, for high throughput apps and data science scripts, it's a worthwhile optimisation.

CHAPTER 4

Schema

The schema is how you define your database tables, columns and relationships.

4.1 Defining a Schema

The schema is usually defined within the `tables.py` file of your Piccolo app (see *Piccolo Apps*).

This reflects the tables in your database. Each table consists of several columns. Here's a very simple schema:

```
# tables.py
from piccolo.table import Table
from piccolo.columns import Varchar


class Band(Table):
    name = Varchar(length=100)
```

For a full list of columns, see *Column Types*.

4.1.1 Default columns

id

Each table is automatically given a `PrimaryKey` column called `id`, which is an auto incrementing integer.

It is used to uniquely identify a row, and is referenced by `ForeignKey` columns on other tables.

If you specify your own `id` column, you may get unexpected behaviour, so it's not recommended at the moment.

4.1.2 Tablename

By default, the name of the table in the database is the Python class name, converted to snakecase. For example `Band -> band`, and `MusicAward -> music_award`.

You can specify a custom tablename to use instead.

```
class Band(Table, tablename="music_band"):
    name = Varchar(length=100)
```

4.1.3 Connecting to the database

In order to create the table and query the database, you need to provide Piccolo with your connection details. See *Engines*.

4.2 Column Types

Hint: You'll notice that the column names tend to match their SQL equivalents.

4.2.1 Column

class piccolo.columns.column_types.**Column**(*null: bool = False*, *primary: bool = False*, *key: bool = False*, *unique: bool = False*, *index: bool = False*, *index_method: piccolo.columns.indexes.IndexMethod = IndexMethod.btree*, *required: bool = False*, *help_text: Optional[str] = None*, *choices: Optional[Type[enum.Enum]] = None*, ***kwargs*)

All other columns inherit from `Column`. Don't use it directly.

The following arguments apply to all column types:

Parameters

- **null** – Whether the column is nullable.
- **primary** – If set, the column is used as a primary key.
- **key** – If set, the column is treated as a key.
- **default** – The column value to use if not specified by the user.
- **unique** – If set, a unique contraint will be added to the column.
- **index** – Whether an index is created for the column, which can improve the speed of selects, but can slow down inserts.
- **index_method** – If index is set to True, this specifies what type of index is created.
- **required** – This isn't used by the database - it's to indicate to other tools that the user must provide this value. Example uses are in serialisers for API endpoints, and form fields.

- **help_text** – This provides some context about what the column is being used for. For example, for a *Decimal* column called *value*, it could say 'The units are millions of dollars'. The database doesn't use this value, but tools such as Piccolo Admin use it to show a tooltip in the GUI.

4.2.2 Bytea

class piccolo.columns.column_types.**Bytea**(*default: Union[bytes, bytearray, enum.Enum, Callable[[], bytes], Callable[[], bytearray], None] = b'', **kwargs*)

Used for storing bytes.

Example

```
class Token(Table):
    token = Bytea(default=b'token123')

# Create
>>> Token(token=b'my-token').save().run_sync()

# Query
>>> Token.select(Token.token).run_sync()
{'token': b'my-token'}
```

Hint: There is also a `Blob` column type, which is an alias for `Bytea`.

4.2.3 Boolean

class piccolo.columns.column_types.**Boolean**(*default: Union[bool, enum.Enum, Callable[[], bool], None] = False, **kwargs*)

Used for storing True / False values. Uses the `bool` type for values.

Example

```
class Band(Table):
    has_drummer = Boolean()

# Create
>>> Band(has_drummer=True).save().run_sync()

# Query
>>> Band.select(Band.has_drummer).run_sync()
{'has_drummer': True}
```

4.2.4 ForeignKey

class piccolo.columns.column_types.**ForeignKey**(*references: t.Union[t.Type[Table], LazyTableReference, str], default: t.Union[int, Enum, None] = None, null: bool = True, on_delete: OnDelete = OnDelete.cascade, on_update: OnUpdate = OnUpdate.cascade, **kwargs*)

Used to reference another table. Uses the `int` type for values.

Example

```
class Band(Table):
    manager = ForeignKey(references=Manager)

# Create
>>> Band(manager=1).save().run_sync()

# Query
>>> Band.select(Band.manager).run_sync()
{'manager': 1}

# Query object
>>> band = await Band.objects().first().run()
>>> band.manager
1
```

Joins

Can also use it to perform joins:

```
>>> await Band.select(Band.name, Band.manager.name).first().run()
{'name': 'Pythonistas', 'manager.name': 'Guido'}
```

To get a referenced row as an object:

```
manager = await Manager.objects().where(
    Manager.id == some_band.manager
).run()
```

Or use either of the following, which are just a proxy to the above:

```
manager = await band.get_related('manager').run()
manager = await band.get_related(Band.manager).run()
```

To change the manager:

```
band.manager = some_manager_id
await band.save().run()
```

Parameters

- **references** – The `Table` being referenced.

  ```
  class Band(Table):
      manager = ForeignKey(references=Manager)
  ```

  A table can have a reference to itself, if you pass a `references` argument of `'self'`.

```
class Musician(Table):
    name = Varchar(length=100)
    instructor = ForeignKey(references='self')
```

In certain situations, you may be unable to reference a `Table` class if it causes a circular dependency. Try and avoid these by refactoring your code. If unavoidable, you can specify a lazy reference. If the `Table` is defined in the same file:

```
class Band(Table):
    manager = ForeignKey(references='Manager')
```

If the `Table` is defined in a Piccolo app:

```
from piccolo.columns.reference import LazyTableReference

class Band(Table):
    manager = ForeignKey(
        references=LazyTableReference(
           table_class_name="Manager", app_name="my_app",
        )
    )
```

If you aren't using Piccolo apps, you can specify a `Table` in any Python module:

```
from piccolo.columns.reference import LazyTableReference

class Band(Table):
    manager = ForeignKey(
        references=LazyTableReference(
           table_class_name="Manager",
           module_path="some_module.tables",
        )
        # Alternatively, Piccolo will interpret this string as
        # the same as above:
        # references="some_module.tables.Manager"
    )
```

- **`on_delete`** – Determines what the database should do when a row is deleted with foreign keys referencing it. If set to `OnDelete.cascade`, any rows referencing the deleted row are also deleted.

  Options:

  – `OnDelete.cascade` (default)

  – `OnDelete.restrict`

  – `OnDelete.no_action`

  – `OnDelete.set_null`

  – `OnDelete.set_default`

  To learn more about the different options, see the Postgres docs.

```
from piccolo.columns import OnDelete

class Band(Table):
    name = ForeignKey(
```

(continues on next page)

(continued from previous page)

```
        references=Manager,
        on_delete=OnDelete.cascade
    )
```

- **on_update** – Determines what the database should do when a row has it's primary key updated. If set to `OnDelete.cascade`, any rows referencing the updated row will have their references updated to point to the new primary key.

  Options:

  – `OnUpdate.cascade` (default)

  – `OnUpdate.restrict`

  – `OnUpdate.no_action`

  – `OnUpdate.set_null`

  – `OnUpdate.set_default`

  To learn more about the different options, see the Postgres docs.

```
from piccolo.columns import OnDelete

class Band(Table):
    name = ForeignKey(
        references=Manager,
        on_update=OnUpdate.cascade
    )
```

4.2.5 Number

BigInt

class piccolo.columns.column_types.**BigInt**(*default: Union[int, enum.Enum, Callable[[], int], None] = 0*, ***kwargs*)

In Postgres, this column supports large integers. In SQLite, it's an alias to an Integer column, which already supports large integers. Uses the `int` type for values.

Example

```
class Band(Table):
    value = BigInt()

# Create
>>> Band(popularity=1000000).save().run_sync()

# Query
>>> Band.select(Band.popularity).run_sync()
{'popularity': 1000000}
```

Integer

class piccolo.columns.column_types.**Integer**(*default: Union[int, enum.Enum, Callable[[], int], None] = 0, **kwargs*)

Used for storing whole numbers. Uses the `int` type for values.

Example

```
class Band(Table):
    popularity = Integer()

# Create
>>> Band(popularity=1000).save().run_sync()

# Query
>>> Band.select(Band.popularity).run_sync()
{'popularity': 1000}
```

Numeric

class piccolo.columns.column_types.**Numeric**(*digits: Optional[Tuple[int, int]] = None, default: Union[decimal.Decimal, enum.Enum, Callable[[], decimal.Decimal], None] = Decimal('0'), **kwargs*)

Used for storing decimal numbers, when precision is important. An example use case is storing financial data. The value is returned as a `Decimal`.

Example

```
from decimal import Decimal

class Ticket(Table):
    price = Numeric(digits=(5,2))

# Create
>>> Ticket(price=Decimal('50.0')).save().run_sync()

# Query
>>> Ticket.select(Ticket.price).run_sync()
{'price': Decimal('50.0')}
```

Parameters **digits** – When creating the column, you specify how many digits are allowed using a tuple. The first value is the *precision*, which is the total number of digits allowed. The second value is the *range*, which specifies how many of those digits are after the decimal point. For example, to store monetary values up to £999.99, the digits argument is *(5,2)*.

Hint: There is also a `Decimal` column type, which is an alias for `Numeric`.

Real

class piccolo.columns.column_types.**Real**(*default: Union[float, enum.Enum, Callable[[], float], None] = 0.0, **kwargs*)

Can be used instead of `Numeric` for storing numbers, when precision isn't as important. The `float` type is

used for values.

Example

```
class Concert(Table):
    rating = Real()

# Create
>>> Concert(rating=7.8).save().run_sync()

# Query
>>> Concert.select(Concert.rating).run_sync()
{'rating': 7.8}
```

Hint: There is also a `Float` column type, which is an alias for `Real`.

SmallInt

class piccolo.columns.column_types.**SmallInt**(*default: Union[int, enum.Enum, Callable[[], int], None] = 0, **kwargs*)

In Postgres, this column supports small integers. In SQLite, it's an alias to an Integer column. Uses the `int` type for values.

Example

```
class Band(Table):
    value = SmallInt()

# Create
>>> Band(popularity=1000).save().run_sync()

# Query
>>> Band.select(Band.popularity).run_sync()
{'popularity': 1000}
```

4.2.6 UUID

class piccolo.columns.column_types.**UUID**(*default: Union[piccolo.columns.defaults.uuid.UUID4, uuid.UUID, enum.Enum, None] = <piccolo.columns.defaults.uuid.UUID4 object>, **kwargs*)

Used for storing UUIDs - in Postgres a UUID column type is used, and in SQLite it's just a Varchar. Uses the `uuid.UUID` type for values.

Example

```
import uuid

class Band(Table):
    uuid = UUID()

# Create
```

(continues on next page)

(continued from previous page)

```
>>> DiscountCode(code=uuid.uuid4()).save().run_sync()

# Query
>>> DiscountCode.select(DiscountCode.code).run_sync()
{'code': UUID('09c4c17d-af68-4ce7-9955-73dcd892e462')}
```

4.2.7 Text

Secret

class piccolo.columns.column_types.**Secret**(*length: int = 255, default: Union[str, enum.Enum, Callable[[], str], None] = '', **kwargs*)

The database treats it the same as a `Varchar`, but Piccolo may treat it differently internally - for example, allowing a user to automatically omit any secret fields when doing a select query, to help prevent inadvertant leakage. A common use for a `Secret` field is a password.

Uses the `str` type for values.

Example

```
class Door(Table):
    code = Secret(length=100)

# Create
>>> Door(code='123abc').save().run_sync()

# Query
>>> Door.select(Door.code).run_sync()
{'code': '123abc'}
```

Text

class piccolo.columns.column_types.**Text**(*default: Union[str, enum.Enum, None, Callable[[], str]] = '', **kwargs*)

Use when you want to store large strings, and don't want to limit the string size. Uses the `str` type for values.

Example

```
class Band(Table):
    name = Text()

# Create
>>> Band(name='Pythonistas').save().run_sync()

# Query
>>> Band.select(Band.name).run_sync()
{'name': 'Pythonistas'}
```

Varchar

class piccolo.columns.column_types.**Varchar**(*length: int = 255, default: Union[str, enum.Enum, Callable[[], str], None] = '', **kwargs*)

Used for storing text when you want to enforce character length limits. Uses the `str` type for values.

Example

```
class Band(Table):
    name = Varchar(length=100)

# Create
>>> Band(name='Pythonistas').save().run_sync()

# Query
>>> Band.select(Band.name).run_sync()
{'name': 'Pythonistas'}
```

Parameters **length** – The maximum number of characters allowed.

4.2.8 Time

Date

class piccolo.columns.column_types.**Date**(*default: Union[piccolo.columns.defaults.date.DateOffset, piccolo.columns.defaults.date.DateCustom, piccolo.columns.defaults.date.DateNow, enum.Enum, None, datetime.date] = <piccolo.columns.defaults.date.DateNow object>, **kwargs*)

Used for storing dates. Uses the `date` type for values.

Example

```
import datetime

class Concert(Table):
    starts = Date()

# Create
>>> Concert(
>>>     starts=datetime.date(year=2020, month=1, day=1)
>>> ).save().run_sync()

# Query
>>> Concert.select(Concert.starts).run_sync()
{'starts': datetime.date(2020, 1, 1)}
```

Interval

class piccolo.columns.column_types.**Interval**(*default: Union[piccolo.columns.defaults.interval.IntervalCustom, enum.Enum, None, datetime.timedelta] = <piccolo.columns.defaults.interval.IntervalCustom object>*, ***kwargs*)

Used for storing timedeltas. Uses the `timedelta` type for values.

Example

```
from datetime import timedelta

class Concert(Table):
    duration = Interval()

# Create
>>> Concert(
>>>     duration=timedelta(hours=2)
>>> ).save().run_sync()

# Query
>>> Concert.select(Concert.duration).run_sync()
{'duration': datetime.timedelta(seconds=7200)}
```

Time

class piccolo.columns.column_types.**Time**(*default: Union[piccolo.columns.defaults.time.TimeCustom, piccolo.columns.defaults.time.TimeNow, piccolo.columns.defaults.time.TimeOffset, enum.Enum, None, datetime.time] = <piccolo.columns.defaults.time.TimeNow object>*, ***kwargs*)

Used for storing times. Uses the `time` type for values.

Example

```
import datetime

class Concert(Table):
    starts = Time()

# Create
>>> Concert(
>>>     starts=datetime.time(hour=20, minute=0, second=0)
>>> ).save().run_sync()

# Query
>>> Concert.select(Concert.starts).run_sync()
{'starts': datetime.time(20, 0, 0)}
```

Timestamp

class piccolo.columns.column_types.**Timestamp**(*default: Union[piccolo.columns.defaults.timestamp.TimestampCusto piccolo.columns.defaults.timestamp.TimestampNow, piccolo.columns.defaults.timestamp.TimestampOffset, enum.Enum, None, datetime.datetime, piccolo.columns.defaults.timestamp.DatetimeDefault] = <piccolo.columns.defaults.timestamp.TimestampNow object>*, ***kwargs*)

Used for storing datetimes. Uses the `datetime` type for values.

Example

```
import datetime

class Concert(Table):
    starts = Timestamp()

# Create
>>> Concert(
>>>     starts=datetime.datetime(year=2050, month=1, day=1)
>>> ).save().run_sync()

# Query
>>> Concert.select(Concert.starts).run_sync()
{'starts': datetime.datetime(2050, 1, 1, 0, 0)}
```

Timestamptz

class piccolo.columns.column_types.**Timestamptz**(*default: Union[piccolo.columns.defaults.timestamptz.Timestampt piccolo.columns.defaults.timestamptz.TimestamptzNow, piccolo.columns.defaults.timestamptz.TimestamptzOffset, enum.Enum, None, datetime.datetime] = <piccolo.columns.defaults.timestamptz.TimestamptzNow object>*, ***kwargs*)

Used for storing timezone aware datetimes. Uses the `datetime` type for values. The values are converted to UTC in the database, and are also returned as UTC.

Example

```
import datetime

class Concert(Table):
    starts = Timestamptz()

# Create
>>> Concert(
>>>     starts=datetime.datetime(
>>>         year=2050, month=1, day=1, tzinfo=datetime.timezone.tz
>>>     )
>>> ).save().run_sync()

# Query
>>> Concert.select(Concert.starts).run_sync()
{
```

(continues on next page)

(continued from previous page)

```
    'starts': datetime.datetime(
        2050, 1, 1, 0, 0, tzinfo=datetime.timezone.utc
    )
}
```

4.2.9 JSON

Storing JSON can be useful in certain situations, for example - raw API responses, data from a Javascript app, and for storing data with an unknown or changing schema.

JSON

class piccolo.columns.column_types.**JSON**(*default: Union[str, List[T], Dict[KT, VT], Callable[[], Union[str, List[T], Dict[KT, VT]]], None] = '{}', **kwargs*)

Used for storing JSON strings. The data is stored as text. This can be preferable to JSONB if you just want to store and retrieve JSON without querying it directly. It works with SQLite and Postgres.

Parameters **default** – Either a JSON string can be provided, or a Python `dict` or `list` which is then converted to a JSON string.

JSONB

class piccolo.columns.column_types.**JSONB**(*default: Union[str, List[T], Dict[KT, VT], Callable[[], Union[str, List[T], Dict[KT, VT]]], None] = '{}', **kwargs*)

Used for storing JSON strings - Postgres only. The data is stored in a binary format, and can be queried. Insertion can be slower (as it needs to be converted to the binary format). The benefits of JSONB generally outweigh the downsides.

Parameters **default** – Either a JSON string can be provided, or a Python `dict` or `list` which is then converted to a JSON string.

arrow

`JSONB` columns have an `arrow` function, which is useful for retrieving a subset of the JSON data, and for filtering in a where clause.

```
# Example schema:
class Booking(Table):
    data = JSONB()

Booking.create_table().run_sync()

# Example data:
Booking.insert(
    Booking(data='{"name": "Alison"}'),
    Booking(data='{"name": "Bob"}')
).run_sync()
```

(continues on next page)

(continued from previous page)

```
# Example queries
>>> Booking.select(
>>>     Booking.id, Booking.data.arrow('name').as_alias('name')
>>> ).run_sync()
[{'id': 1, 'name': '"Alison"'}, {'id': 2, 'name': '"Bob"'}]

>>> Booking.select(Booking.id).where(
>>>     Booking.data.arrow('name') == '"Alison"'
>>> ).run_sync()
[{'id': 1}]
```

4.2.10 Array

Arrays of data can be stored, which can be useful when you want store lots of values without using foreign keys.

class piccolo.columns.column_types.**Array**(*base_column: piccolo.columns.base.Column, default: Union[List[T], enum.Enum, Callable[[], List[T]], None] = <class 'list'>, **kwargs*)

Used for storing lists of data.

Example

```
class Ticket(Table):
    seat_numbers = Array(base_column=Integer())

# Create
>>> Ticket(seat_numbers=[34, 35, 36]).save().run_sync()

# Query
>>> Ticket.select(Ticket.seat_numbers).run_sync()
{'seat_numbers': [34, 35, 36]}
```

Accessing individual elements

Array.**__getitem__**(*value: int*) → piccolo.columns.column_types.Array

Allows queries which retrieve an item from the array. The index starts with 0 for the first value. If you were to write the SQL by hand, the first index would be 1 instead:

https://www.postgresql.org/docs/current/arrays.html

However, we keep the first index as 0 to fit better with Python.

For example:

```
>>> Ticket.select(Ticket.seat_numbers[0]).first().run_sync
{'seat_numbers': 325}
```

any

Array.**any**(*value: Any*) → piccolo.columns.combination.Where

Check if any of the items in the array match the given value.

```
>>> Ticket.select().where(Ticket.seat_numbers.any(510)).run_sync()
```

all

`Array.all(value: Any)` → piccolo.columns.combination.Where

Check if all of the items in the array match the given value.

```
>>> Ticket.select().where(Ticket.seat_numbers.all(510)).run_sync()
```

4.3 Advanced

4.3.1 Readable

Sometimes Piccolo needs a succinct representation of a row - for example, when displaying a link in the Piccolo Admin GUI (see *Ecosystem*). Rather than just displaying the row ID, we can specify something more user friendly using `Readable`.

```
# tables.py
from piccolo.table import Table
from piccolo.columns import Varchar
from piccolo.columns.readable import Readable


class Band(Table, tablename="music_band"):
    name = Varchar(length=100)

    @classmethod
    def get_readable(cls):
        return Readable(template="%s", columns=[cls.name])
```

Specifying the `get_readable` classmethod isn't just beneficial for Piccolo tooling - you can also use it your own queries.

```
Band.select(Band.get_readable()).run_sync()
```

Here is an example of a more complex `Readable`.

```
class Band(Table, tablename="music_band"):
    name = Varchar(length=100)

    @classmethod
    def get_readable(cls):
        return Readable(template="Band %s - %s", columns=[cls.id, cls.name])
```

As you can see, the template can include multiple columns, and can contain your own text.

4.3.2 Table Tags

`Table` subclasses can be given tags. The tags can be used for filtering, for example with `table_finder` (see *table_finder*).

```
class Band(Table, tags=["music"]):
    name = Varchar(length=100)
```

4.3.3 Mixins

If you're frequently defining the same columns over and over again, you can use mixins to reduce the amount of repetition.

```
from piccolo.columns import Varchar, Boolean
from piccolo.table import Table


class FavouriteMixin:
    favourite = Boolean(default=False)


class Manager(FavouriteMixin, Table):
    name = Varchar()
```

4.3.4 Choices

You can specify choices for a column, using Python's `Enum` support.

```
from enum import Enum

from piccolo.columns import Varchar
from piccolo.table import Table


class Shirt(Table):
    class Size(str, Enum):
        small = 's'
        medium = 'm'
        large = 'l'

    size = Varchar(length=1, choices=Size)
```

We can then use the `Enum` in our queries.

```
>>> Shirt(size=Shirt.Size.large).save().run_sync()

>>> Shirt.select().run_sync()
[{'id': 1, 'size': 'l'}]
```

Note how the value stored in the database is the `Enum` value (in this case `'l'`).

You can also use the `Enum` in `where` clauses, and in most other situations where a query requires a value.

```
>>> Shirt.insert(
>>>     Shirt(size=Shirt.Size.small),
>>>     Shirt(size=Shirt.Size.medium)
>>> ).run_sync()

>>> Shirt.select().where(Shirt.size == Shirt.Size.small).run_sync()
[{'id': 1, 'size': 's'}]
```

Advantages

By using choices, you get the following benefits:

- Signalling to other programmers what values are acceptable for the column.
- Improved storage efficiency (we can store `'l'` instead of `'large'`).
- Piccolo Admin support

CHAPTER 5

Projects and Apps

By using Piccolo projects and apps, you can build a larger, more modular, application.

5.1 Piccolo Projects

A Piccolo project is a collection of apps.

5.1.1 piccolo_conf.py

A project requires a *piccolo_conf.py* file. To create this file, use the following command:

```
piccolo project new
```

The file serves two important purposes:

- Contains your database settings
- Is used for registering *Piccolo Apps*.

5.1.2 Example

Here's an example:

```
from piccolo.engine.postgres import PostgresEngine


from piccolo.conf.apps import AppRegistry
```

(continues on next page)

(continued from previous page)

```
DB = PostgresEngine(
    config={
        "database": "piccolo_project",
        "user": "postgres",
        "password": "",
        "host": "localhost",
        "port": 5432,
    }
)


APP_REGISTRY = AppRegistry(
    apps=["home.piccolo_app", "piccolo_admin.piccolo_app"]
)
```

5.1.3 DB

The DB setting is an `Engine` instance. To learn more Engines, see *Engines*.

5.1.4 APP_REGISTRY

The `APP_REGISTRY` setting is an `AppRegistry` instance.

class piccolo.conf.apps.**AppRegistry**(*apps: List[str] = <factory>*)

Records all of the Piccolo apps in your project. Kept in piccolo_conf.py.

Parameters **apps** – A list of paths to Piccolo apps, e.g. [‘blog.piccolo_app’]

5.2 Piccolo Apps

By leveraging Piccolo apps you can:

- Modularise your code.
- Share your apps with other Piccolo users.
- Unlock some useful functionality like auto migrations.

5.2.1 Creating an app

Run the following command within your project:

```
piccolo app new my_app
```

Where *my_app* is your new app’s name. This will create a folder like this:

```
my_app/
    __init__.py
    piccolo_app.py
    piccolo_migrations/
        __init__.py
    tables.py
```

It's important to register your new app with the `APP_REGISTRY` in *piccolo_conf.py*.

```
# piccolo_conf.py
APP_REGISTRY = AppRegistry(apps=['my_app.piccolo_app'])
```

Anytime you invoke the *piccolo* command, you will now be able to perform operations on your app, such as *Migrations*.

5.2.2 AppConfig

Inside your app's *piccolo_app.py* file is an `AppConfig` instance. This is how you customise your app's settings.

```
# piccolo_app.py
import os

from piccolo.conf.apps import AppConfig
from .tables import (
    Author,
    Post,
    Category,
    CategoryToPost,
)


CURRENT_DIRECTORY = os.path.dirname(os.path.abspath(__file__))


APP_CONFIG = AppConfig(
    app_name='blog',
    migrations_folder_path=os.path.join(CURRENT_DIRECTORY, 'piccolo_migrations'),
    table_classes=[Author, Post, Category, CategoryToPost],
    migration_dependencies=[],
    commands=[]
)
```

app_name

This is used to identify your app, when using the *piccolo* CLI, for example:

```
piccolo migrations forwards blog
```

migrations_folder_path

Specifies where your app's migrations are stored. By default, a folder called *piccolo_migrations* is used.

table_classes

Use this to register your app's `Table` subclasses. This is important for auto migrations (see *Migrations*).

You can register them manually, see the example above, or can use `table_finder`.

table_finder

Instead of manually registering `Table` subclasses, you can use `table_finder` to automatically import any `Table` subclasses from a given list of modules.

```
from piccolo.conf.apps import table_finder

APP_CONFIG = AppConfig(
    app_name='blog',
    migrations_folder_path=os.path.join(CURRENT_DIRECTORY, 'piccolo_migrations'),
    table_classes=table_finder(modules=['blog.tables']),
    migration_dependencies=[],
    commands=[]
)
```

The module path should be from the root of the project (the same directory as your `piccolo_conf.py` file, rather than a relative path).

You can filter the `Table` subclasses returned using tags (see *Table Tags*).

piccolo.conf.apps.**table_finder**(*modules: Sequence[str]*, *include_tags: Sequence[str] = ['__all__']*, *exclude_tags: Sequence[str] = []*) → List[Type[piccolo.table.Table]]

Rather than explicitly importing and registering table classes with the AppConfig, `table_finder` can be used instead. It imports any `Table` subclasses in the given modules. Tags can be used to limit which `Table` subclasses are imported.

Parameters

- **modules** – The module paths to check for `Table` subclasses. For example, ['blog.tables']. The path should be from the root of your project, not a relative path.
- **include_tags** – If the `Table` subclass has one of these tags, it will be imported. The special tag '__all__' will import all `Table` subclasses found.
- **exclude_tags** – If the `Table` subclass has any of these tags, it won't be imported. *exclude_tags* overrides *include_tags*.

migration_dependencies

Used to specify other Piccolo apps whose migrations need to be run before the current app's migrations.

commands

You can register functions and coroutines, which are automatically added to the *piccolo* CLI.

The targ library is used under the hood. It makes it really easy to write command lines tools - just use type annotations and docstrings. Here's an example:

```
def say_hello(name: str):
    """
    Say hello.

    :param name:
        The person to greet.

    """
    print(name)
```

We then register it with the `AppConfig`.

```
# piccolo_app.py

APP_CONFIG = AppConfig(
    # ...
    commands=[say_hello]
)
```

And from the command line:

```
>>> piccolo my_app say_hello bob
bob
```

By convention, store the command definitions in a *commands* folder in your app.

```
my_app/
    __init__.py
    piccolo_app.py
    commands/
        __init__.py
        say_hello.py
```

Piccolo itself is bundled with several apps - have a look at the source code for inspiration.

5.2.3 Sharing Apps

By breaking up your project into apps, the project becomes more maintainable. You can also share these apps between projects, and they can even be installed using pip.

5.3 Included Apps

Just as you can modularise your own code using *apps*, Piccolo itself ships with several builtin apps, which provide a lot of its functionality.

5.3.1 Auto includes

The following are registered with your *AppRegistry* automatically:

app

Lets you create new Piccolo apps. See *Piccolo Apps*.

```
piccolo app new
```

asgi

Lets you scaffold an ASGI web app. See *ASGI*.

```
piccolo asgi new
```

meta

Tells you which version of Piccolo is installed.

```
piccolo meta version
```

migrations

Lets you create and run migrations. See *Migrations*.

playground

Lets you learn the Piccolo query syntax, using an example schema. See *Playground*.

```
piccolo playground run
```

project

Lets you create a new `piccolo_conf.py` file. See *Piccolo Projects*.

```
piccolo project new
```

shell

Launches an iPython shell, and automatically imports all of your registered `Table` classes. It's great for running adhoc database queries using Piccolo.

```
piccolo shell run
```

sql_shell

Launches a SQL shell (`psql` or `sqlite3` depending on the engine), using the connection settings defined in `piccolo_conf.py`. It's convenient if you need to run raw SQL queries on your database.

```
piccolo sql_shell run
```

For it to work, the underlying command needs to be on the path (i.e. `psql` or `sqlite3` depending on which you're using).

5.3.2 Optional includes

These need to be explicitly registered with your *AppRegistry*.

user

Provides a user table, and commands for creating / managing users. See *Authentication*.

CHAPTER 6

Engines

Engines are what execute the SQL queries. Each supported backend has its own engine (see *Engine types*).

It's important that each `Table` class knows which engine to use. There are two ways of doing this - setting it explicitly via the `db` argument, or letting Piccolo find it using `engine_finder`.

6.1 Explicit

This can be useful when writing a simple script which needs to use Piccolo to connect to a database.

```
from piccolo.engine.sqlite import SQLiteEngine
from piccolo.table import Table
from piccolo.columns import Varchar


DB = SQLiteEngine(path='my_db.sqlite')


# Here we explicitly reference an engine:
class MyTable(Table, db=DB):
    name = Varchar()
```

6.2 engine_finder

By default Piccolo uses `engine_finder`. Piccolo will look for a file called `piccolo_conf.py` on the path, and will try and import a `DB` variable, which defines the engine.

You can ask Piccolo to create the `piccolo_conf.py` file for you, using the following command:

```
piccolo project new
```

Here's an example `piccolo_conf.py` file:

```
# piccolo_conf.py
from piccolo.engine.sqlite import SQLiteEngine


DB = SQLiteEngine(path='my_db.sqlite')
```

Hint: A good place for your piccolo_conf file is at the root of your project, where the Python interpreter will be launched.

6.2.1 PICCOLO_CONF environment variable

You can modify the configuration file location by using the `PICCOLO_CONF` environment variable.

In your terminal:

```
export PICCOLO_CONF=piccolo_conf_test
```

Or at the entypoint for your app, before any other imports:

```
import os
os.environ['PICCOLO_CONF'] = 'piccolo_conf_test'
```

This is helpful during tests - you can specify a different configuration file which contains the connection details for a test database. Similarly, it's useful if you're deploying your code to different environments (e.g. staging and production). Have two configuration files, and set the environment variable accordingly.

```
# An example piccolo_conf_test.py
from piccolo.engine.sqlite import SQLiteEngine


DB = SQLiteEngine(path='my_test_db.sqlite')
```

Hint: You can also specify sub modules, like *my_module.piccolo_conf*.

6.3 Engine types

Hint: Postgres is the preferred database to use, especially in production. It is the most feature complete.

6.3.1 SQLiteEngine

Configuration

The `SQLiteEngine` is very simple - just specify a file path. The database file will be created automatically if it doesn't exist.

```
# piccolo_conf.py
from piccolo.engine.sqlite import SQLiteEngine


DB = SQLiteEngine(path='my_app.sqlite')
```

Source

class piccolo.engine.sqlite.**SQLiteEngine**(*path: str = 'piccolo.sqlite'*, *detect_types=1*, *isolation_level=None*, ***connection_kwargs*)

Any connection kwargs are passed into the database adapter.

See here for more info: https://docs.python.org/3/library/sqlite3.html#sqlite3.connect

6.3.2 PostgresEngine

Configuration

```
# piccolo_conf.py
from piccolo.engine.postgres import PostgresEngine


DB = PostgresEngine(config={
    'host': 'localhost',
    'database': 'my_app',
    'user': 'postgres',
    'password': ''
})
```

config

The config dictionary is passed directly to the underlying database adapter, asyncpg. See the asyncpg docs to learn more.

Connection pool

To use a connection pool, you need to first initialise it. The best place to do this is in the startup event handler of whichever web framework you are using.

Here's an example using Starlette. Notice that we also close the connection pool in the shutdown event handler.

```
from piccolo.engine import engine_finder
from starlette.applications import Starlette


app = Starlette()


@app.on_event('startup')
async def open_database_connection_pool():
    engine = engine_finder()
    await engine.start_connection_pool()


@app.on_event('shutdown')
async def close_database_connection_pool():
    engine = engine_finder()
    await engine.close_connection_pool()
```

Hint: Using a connection pool helps with performance, since connections are reused instead of being created for each query.

Once a connection pool has been started, the engine will use it for making queries.

Hint: If you're running several instances of an app on the same server, you may prefer an external connection pooler - like pgbouncer.

Configuration

The connection pool uses the same configuration as your engine. You can also pass in additional parameters, which are passed to the underlying database adapter. Here's an example:

```
# To increase the number of connections available:
await engine.start_connection_pool(max_size=20)
```

Source

class piccolo.engine.postgres.**PostgresEngine**(*config: Dict[str, Any]*, *extensions: Sequence[str] = ['uuid-ossp']*, *log_queries: bool = False*)

Used to connect to Postgresql.

Parameters

- **config** – The config dictionary is passed to the underlying database adapter, asyncpg. Common arguments you're likely to need are:
  - host
  - port
  - user

  - password
  - database

  For example, `{'host': 'localhost', 'port': 5432}`.

  To see all available options:

  - https://magicstack.github.io/asyncpg/current/api/index.html#connection

- **`extensions`** – When the engine starts, it will try and create these extensions in Postgres.
- **`log_queries`** – If True, all SQL and DDL statements are printed out before being run. Useful for debugging.

CHAPTER 7

Migrations

7.1 Creating migrations

Migrations are Python files which are used to modify the database schema in a controlled way. Each migration belongs to a Piccolo app (see *Piccolo Apps*).

You can either manually populate migrations, or allow Piccolo to do it for you automatically. To create an empty migration:

```
piccolo migrations new my_app
```

This creates a new migration file in the migrations folder of the app. The migration filename is a timestamp, which also serves as the migration ID.

```
piccolo_migrations/
    2018-09-04T19:44:09.py
```

The contents of an empty migration file looks like this:

```
from piccolo.apps.migrations.auto import MigrationManager


ID = '2018-09-04T19:44:09'


async def forwards():
    manager = MigrationManager(migration_id=ID)

    def run():
        print(f"running {ID}")

    manager.add_raw(run)
    return manager
```

Replace the *run* function with whatever you want the migration to do - typically running some SQL. It can be a function or a coroutine.

7.1.1 The golden rule

Never import your tables directly into a migration, and run methods on them.

This is a **bad example**:

```
from ..tables import Band

ID = '2018-09-04T19:44:09'


async def forwards():
    manager = MigrationManager(migration_id=ID)

    async def run():
        await Band.create_table().run()

    manager.add_raw(run)
    return manager
```

The reason you don't want to do this, is your tables will change over time. If someone runs your migrations in the future, they will get different results. Make your migrations completely independent of other code, so they're self contained and repeatable.

7.1.2 Auto migrations

Manually writing your migrations gives you a good level of control, but Piccolo supports *auto migrations* which can save a great deal of time.

Piccolo will work out which tables to add by comparing previous auto migrations, and your current tables. In order for this to work, you have to register your app's tables with the *AppConfig* in the piccolo_app.py file at the root of your app (see *Piccolo Apps*).

Creating an auto migration:

```
piccolo migrations new my_app --auto
```

Hint: Auto migrations are the preferred way to create migrations with Piccolo. We recommend using *empty migrations* for special circumstances which aren't supported by auto migrations, or to modify the data held in tables, as opposed to changing the tables themselves.

Warning: Auto migrations aren't supported in SQLite, because of SQLite's extremely limited support for SQL Alter statements. This might change in the future.

Troubleshooting

Auto migrations can accommodate most schema changes. There may be some rare edge cases where a single migration is trying to do too much in one go, and fails. To avoid these situations, create auto migrations frequently, and keep

them fairly small.

7.2 Running migrations

Hint: To see all available options for these commands, use the `--help` flag, for example `piccolo migrations forwards --help`.

7.2.1 Forwards

When the migration is run, the forwards function is executed. To do this:

```
piccolo migrations forwards my_app
```

7.2.2 Reversing migrations

To reverse the migration, run this:

```
piccolo migrations backwards 2018-09-04T19:44:09
```

You can try going forwards and backwards a few times to make sure it works as expected.

7.2.3 Checking migrations

You can easily check which migrations have and haven't ran using the following:

```
piccolo migrations check
```

CHAPTER 8

Authentication

Piccolo ships with some authentication support out of the box.

8.1 Registering the app

Make sure `'piccolo.apps.user.piccolo_app'` is in your `AppRegistry` (see *Piccolo Projects*).

8.2 Tables

8.2.1 BaseUser

`BaseUser` is a `Table` you can use to store and authenticate your users.

Creating the Table

Run the migrations:

```
piccolo migrations forwards user
```

Commands

The app comes with some useful commands.

user create

Create a new user.

```
piccolo user create
```

user change_password

Change a user's password.

```
piccolo user change_password
```

user change_permissions

Change a user's permissions. The options are `--admin`, `--superuser` and `--active`, which change the corresponding attributes on `BaseUser`.

For example:

```
piccolo user change_permissions some_user --active=true
```

The Piccolo Admin (see *Ecosystem*) uses these attributes to control who can login and what they can do.

- **active** and **admin** - must be true for a user to be able to login.
- **superuser** - must be true for a user to be able to change other user's passwords.

Within your code

login

To check a user's credentials, do the following:

```
from piccolo.apps.user.tables import BaseUser

# From within a coroutine:
>>> await BaseUser.login(username="bob", password="abc123")
1

# When not in an event loop:
>>> BaseUser.login_sync(username="bob", password="abc123")
1
```

If the login is successful, the user's id is returned, otherwise `None` is returned.

update_password / update_password_sync

To change a user's password:

```
# From within a coroutine:
await BaseUser.update_password(username="bob", password="abc123")

# When not in an event loop:
BaseUser.update_password_sync(username="bob", password="abc123")
```

Warning: Don't use bulk updates for passwords - use `update_password` / `update_password_sync`, and they'll correctly hash the password.

CHAPTER 9

ASGI

Using Piccolo standalone is fine if you want to build a data science script, but often you'll want to build a web application around it.

ASGI is a standardised way for async Python libraries to interoperate. It's the equivalent of WSGI in the synchronous world.

By using the `piccolo asgi new` command, Piccolo will scaffold an ASGI web app for you, which includes everything you need to get started. The command will ask for your preferences on which libraries to use.

9.1 Routing frameworks

Currently, Starlette, FastAPI, and BlackSheep are supported.

Other great ASGI routing frameworks exist, and may be supported in the future (Quart , Sanic , Django etc).

9.1.1 Which to use?

All are great choices. FastAPI is built on top of Starlette, so they're very similar. FastAPI is useful if you want to document a REST API.

9.2 Web servers

Hypercorn and Uvicorn are available as ASGI servers. Daphne can't be used programatically so was omitted at this time.

CHAPTER 10

Features

10.1 Tab Completion

Piccolo does everything possible to support tab completion. It has been tested with iPython and VSCode.

To find out more about how it was done, read this article.

10.2 Supported Databases

10.2.1 Postgres

Postgres is the primary focus for Piccolo, and is what we expect most people will be using in production.

10.2.2 SQLite

SQLite support is not as complete as Postgres, but it is available - mostly because it's easy to setup.

10.3 Security

10.3.1 SQL Injection protection

If you look under the hood, Piccolo uses a custom class called *QueryString* for composing queries. It keeps query parameters separate from the query string, so we can pass parameterised queries to the engine. This helps prevent SQL Injection attacks.

10.4 Syntax

10.4.1 As close as possible to SQL

The classes / methods / functions in Piccolo mirror their SQL counterparts as closely as possible.

For example:

- In other ORMs, you define models - in Piccolo you define tables.
- Rather than using a filter method, you use a *where* method like in SQL.

10.4.2 Get the SQL at any time

At any time you can access the `__str__` method of a query, to see the underlying SQL - making the ORM feel less magic.

```
>>> query = Band.select(Band.name).where(Band.popularity >= 100)
>>> print(query)
'SELECT name from band where popularity > 100'
```

CHAPTER 11

Playground

Piccolo ships with a handy command to help learn the different queries. For simple usage see *Playground*.

11.1 Advanced Playground Usage

11.1.1 Postgres

If you want to use Postgres instead of SQLite, you need to create a database first.

Install Postgres

See *Setup Postgres*.

Create database

By default the playground expects a local database to exist with the following credentials:

```
user: "piccolo"
password: "piccolo"
host: "localhost"  # or 127.0.0.1
database: "piccolo_playground"
port: 5432
```

You can create a database using pgAdmin.

If you want to use different credentials, you can pass them into the playground command (use `piccolo playground run --help` for details).

Connecting

When you have the database setup, you can connect to it as follows:

```
piccolo playground run --engine=postgres
```

CHAPTER 12

Deployment

12.1 Docker

Piccolo has several dependencies which are compiled (e.g. asyncpg, orjson), which is great for performance, but you may run into difficulties when using Alpine Linux as your base Docker image.

Alpine uses a different compiler toolchain to most Linux distros. It's highly recommended to use Debian as your base Docker image. Many Python packages have prebuilt versions for Debian, meaning you don't have to compile them at all during install. The result is a much faster build process, and potentially even a smaller overall Docker image size (the size of Alpine quickly balloons after you've added all of the compilation dependencies).

CHAPTER 13

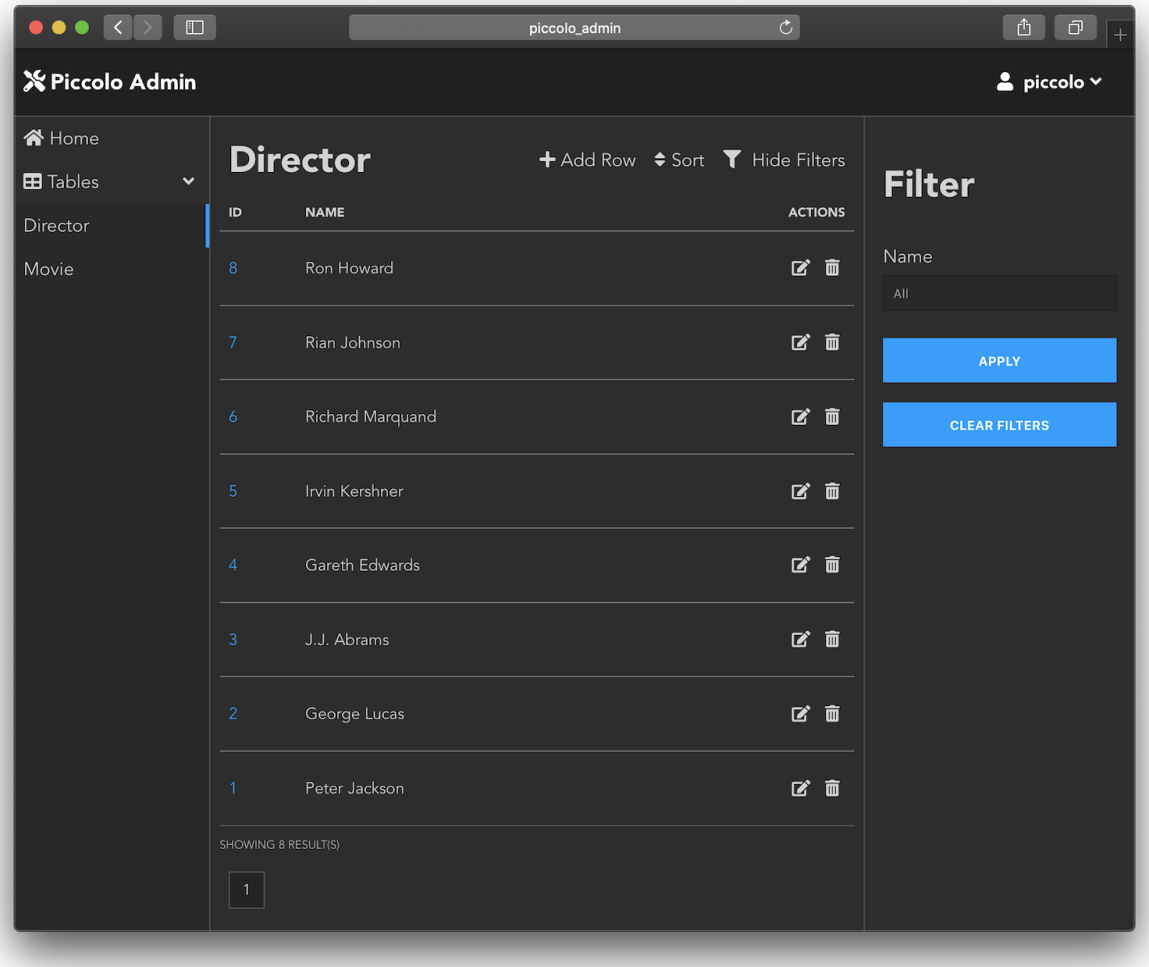
Ecosystem

13.1 Piccolo API

Provides some handy utilities for creating an API around your Piccolo tables. Examples include easy CRUD endpoints for ASGI apps, authentication and rate limiting. Read the docs.

13.2 Piccolo Admin

Lets you create a powerful web GUI for your tables in two minutes. View the project on Github.



13.3 Piccolo Examples

A repository containing example projects built with Piccolo, as well as links to community projects.

CHAPTER 14

Contributing

If you want to dig deeper into the Piccolo internals, follow these instructions.

14.1 Get the tests running

- Create a new virtualenv
- Clone the Git repo.
- Install the dependencies: `pip install -r requirements.txt`.
- `cd tests`
- Install the test dependencies: `pip install -r test-requirements.txt`.
- Setup Postgres
- Run the tests: `./run-tests.sh`

14.2 Contributing to the docs

The docs are written using Sphinx. To get them running locally:

- `cd docs`
- Install the requirements: `pip install -r doc-requirements.txt`
- Do an initial build of the docs: `make html`
- Serve the docs: `python serve_docs.py`
- The docs will auto rebuild as you make changes.

14.3 Code style

Piccolo uses Black for formatting, preferably with a max line length of 79, to keep it consistent with PEP8 .

You can configure VSCode by modifying `settings.json` as follows:

```
{
    "python.linting.enabled": true,
    "python.linting.mypyEnabled": true,
    "python.formatting.provider": "black",
    "python.formatting.blackArgs": [
        "--line-length",
        "79"
    ],
    "editor.formatOnSave": true
}
```

Type hints are used throughout the project.

CHAPTER 15

Changes

15.1 0.21.2

Fixing a bug with serialising `Enum` instances in migrations. For example: `Varchar(default=Colour.red)`.

15.2 0.21.1

Fix missing imports in FastAPI and Starlette app templates.

15.3 0.21.0

- Added a `freeze` method to `Query`.
- Added BlackSheep as an option to `piccolo asgi new`.

15.4 0.20.0

Added `choices` option to `Column`.

15.5 0.19.1

- Added `piccolo user change_permissions` command.
- Added aliases for CLI commands.

15.6 0.19.0

Changes to the `BaseUser` table - added a `superuser`, and `last_login` column. These are required for upgrades to Piccolo Admin.

If you're using migrations, then running `piccolo migrations forwards all` should add these new columns for you.

If not using migrations, the `BaseUser` table can be upgraded using the following DDL statements:

```
ALTER TABLE piccolo_user ADD COLUMN "superuser" BOOLEAN NOT NULL DEFAULT false
ALTER TABLE piccolo_user ADD COLUMN "last_login" TIMESTAMP DEFAULT null
```

15.7 0.18.4

- Fixed a bug when multiple tables inherit from the same mixin (thanks to @brnosouza).
- Added a `log_queries` option to `PostgresEngine`, which is useful during debugging.
- Added the *inflection* library for converting `Table` class names to database table names. Previously, a class called `TableA` would wrongly have a table called `table` instead of `table_a`.
- Fixed a bug with `SerialisedBuiltin.__hash__` not returning a number, which could break migrations (thanks to @sinisaos).

15.8 0.18.3

Improved `Array` column serialisation - needed to fix auto migrations.

15.9 0.18.2

Added support for filtering `Array` columns.

15.10 0.18.1

Add the `Array` column type as a top level import in `piccolo.columns`.

15.11 0.18.0

- Refactored `forwards` and `backwards` commands for migrations, to make them easier to run programatically.
- Added a simple `Array` column type.
- `table_finder` now works if just a string is passed in, instead of having to pass in an array of strings.

15.12 0.17.5

Catching database connection exceptions when starting the default ASGI app created with `piccolo asgi new` - these errors exist if the Postgres database hasn't been created yet.

15.13 0.17.4

Added a `help_text` option to the `Table` metaclass. This is used in Piccolo Admin to show tooltips.

15.14 0.17.3

Added a `help_text` option to the `Column` constructor. This is used in Piccolo Admin to show tooltips.

15.15 0.17.2

- Exposing `index_type` in the `Column` constructor.
- Fixing a typo with `start_connection_pool` and ``close_connection_pool` - thanks to paolodina for finding this.
- Fixing a typo in the `PostgresEngine` docs - courtesy of paolodina.

15.16 0.17.1

Fixing a bug with `SchemaSnapshot` if column types were changed in migrations - the snapshot didn't reflect the changes.

15.17 0.17.0

- Migrations now directly import `Column` classes - this allows users to create custom `Column` subclasses. Migrations previously only worked with the builtin column types.
- Migrations now detect if the column type has changed, and will try and convert it automatically.

15.18 0.16.5

The Postgres extensions that `PostgresEngine` tries to enable at startup can now be configured.

15.19 0.16.4

- Fixed a bug with `MyTable.column != None`
- Added `is_null` and `is_not_null` methods, to avoid linting issues when comparing with None.

15.20 0.16.3

- Added `WhereRaw`, so raw SQL can be used in where clauses.
- `piccolo shell run` now uses syntax highlighting - courtesy of Fingel.

15.21 0.16.2

Reordering the dependencies in requirements.txt when using `piccolo asgi new` as the latest FastAPI and Starlette versions are incompatible.

15.22 0.16.1

Added `Timestamptz` column type, for storing datetimes which are timezone aware.

15.23 0.16.0

- Fixed a bug with creating a `ForeignKey` column with `references="self"` in auto migrations.
- Changed migration file naming, so there are no characters in there which are unsupported on Windows.

15.24 0.15.1

Changing the status code when creating a migration, and no changes were detected. It now returns a status code of 0, so it doesn't fail build scripts.

15.25 0.15.0

Added `Bytea` / `Blob` column type.

15.26 0.14.13

Fixing a bug with migrations which drop column defaults.

15.27 0.14.12

- Fixing a bug where re-running `Table.create(if_not_exists=True)` would fail if it contained columns with indexes.
- Raising a `ValueError` if a relative path is provided to `ForeignKey references`. For example, `.tables.Manager`. The paths must be absolute for now.

15.28 0.14.11

Fixing a bug with `Boolean` column defaults, caused by the `Table` metaclass not being explicit enough when checking falsy values.

15.29 0.14.10

- The `ForeignKey references` argument can now be specified using a string, or a `LazyTableReference` instance, rather than just a `Table` subclass. This allows a `Table` to be specified which is in a Piccolo app, or Python module. The `Table` is only loaded after imports have completed, which prevents circular import issues.
- Faster column copying, which is important when specifying joins, e.g. `await Band.select(Band.manager.name).run()`.
- Fixed a bug with migrations and foreign key contraints.

15.30 0.14.9

Modified the exit codes for the `forwards` and `backwards` commands when no migrations are left to run / reverse. Otherwise build scripts may fail.

15.31 0.14.8

- Improved the method signature of the `output` query clause (explicitly added args, instead of using `**kwargs`).
- Fixed a bug where `output(as_list=True)` would fail if no rows were found.
- Made `piccolo migrations forwards` command output more legible.
- Improved renamed table detection in migrations.
- Added the `piccolo migrations clean` command for removing orphaned rows from the migrations table.
- Fixed a bug where `get_migration_managers` wasn't inclusive.
- Raising a `ValueError` if `is_in` or `not_in` query clauses are passed an empty list.
- Changed the migration commands to be top level async.
- Combined `print` and `sys.exit` statements.

15.32 0.14.7

- Added missing type annotation for `run_sync`.
- Updating type annotations for column default values - allowing callables.
- Replaced instances of `asyncio.run` with `run_sync`.
- Tidied up aiosqlite imports.

15.33 0.14.6

- Added JSON and JSONB column types, and the arrow function for JSONB.
- Fixed a bug with the distinct clause.
- Added `as_alias`, so select queries can override column names in the response (i.e. SELECT foo AS bar from baz).
- Refactored JSON encoding into a separate utils file.

15.34 0.14.5

- Removed old iPython version recommendation in the `piccolo shell run` and `piccolo playground run`, and enabled top level await.
- Fixing outstanding mypy warnings.
- Added optional requirements for the playground to setup.py

15.35 0.14.4

- Added `piccolo sql_shell run` command, which launches the psql or sqlite3 shell, using the connection parameters defined in `piccolo_conf.py`. This is convenient when you want to run raw SQL on your database.
- `run_sync` now handles more edge cases, for example if there's already an event loop in the current thread.
- Removed asgiref dependency.

15.36 0.14.3

- Queries can be directly awaited - `await MyTable.select()`, as an alternative to using the run method `await MyTable.select().run()`.
- The `piccolo asgi new` command now accepts a `name` argument, which is used to populate the default database name within the template.

15.37 0.14.2

- Centralised code for importing Piccolo apps and tables - laying the foundation for fixtures.
- Made orjson an optional dependency, installable using `pip install piccolo[orjson]`.
- Improved version number parsing in Postgres.

15.38 0.14.1

Fixing a bug with dropping tables in auto migrations.

15.39 0.14.0

Added `Interval` column type.

15.40 0.13.5

- Added `allowed_hosts` to `create_admin` in ASGI template.
- Fixing bug with default `root` argument in some piccolo commands.

15.41 0.13.4

- Fixed bug with `SchemaSnapshot` when dropping columns.
- Added custom `__repr__` method to `Table`.

15.42 0.13.3

Added `piccolo shell run` command for running adhoc queries using Piccolo.

15.43 0.13.2

- Fixing bug with auto migrations when dropping columns.
- Added a `root` argument to `piccolo asgi new`, `piccolo app new` and `piccolo project new` commands, to override where the files are placed.

15.44 0.13.1

Added support for `group_by` and `Count` for aggregate queries.

15.45 0.13.0

Added *required* argument to `Column`. This allows the user to indicate which fields must be provided by the user. Other tools can use this value when generating forms and serialisers.

15.46 0.12.6

- Fixing a typo in `TimestampCustom` arguments.
- Fixing bug in `TimestampCustom` SQL representation.
- Added more extensive deserialisation for migrations.

15.47 0.12.5

- Improved `PostgresEngine` docstring.
- Resolving rename migrations before adding columns.
- Fixed bug serialising `TimestampCustom`.
- Fixed bug with altering column defaults to be non-static values.
- Removed `response_handler` from `Alter` query.

15.48 0.12.4

Using orjson for JSON serialisation when using the `output(as_json=True)` clause. It supports more Python types than ujson.

15.49 0.12.3

Improved `piccolo user create` command - defaults the username to the current system user.

15.50 0.12.2

Fixing bug when sorting `extra_definitions` in auto migrations.

15.51 0.12.1

- Fixed typos.
- Bumped requirements.

15.52 0.12.0

- Added `Date` and `Time` columns.
- Improved support for column default values.
- Auto migrations can now serialise more Python types.
- Added `Table.indexes` method for listing table indexes.
- Auto migrations can handle adding / removing indexes.
- Improved ASGI template for FastAPI.

15.53 0.11.8

ASGI template fix.

15.54 0.11.7

- Improved `UUID` columns in SQLite - prepending 'uuid:' to the stored value to make the type more explicit for the engine.
- Removed SQLite as an option for `piccolo asgi new` until auto migrations are supported.

15.55 0.11.6

Added support for FastAPI to `piccolo asgi new`.

15.56 0.11.5

Fixed bug in `BaseMigrationManager.get_migration_modules` - wasn't excluding non-Python files well enough.

15.57 0.11.4

- Stopped `piccolo migrations new` from creating a config.py file - was legacy.
- Added a README file to the *piccolo_migrations* folder in the ASGI template.

15.58 0.11.3

Fixed *__pycache__* bug when using `piccolo asgi new`.

15.59 0.11.2

- Showing a warning if trying auto migrations with SQLite.
- Added a command for creating a new ASGI app - `piccolo asgi new`.
- Added a meta app for printing out the Piccolo version - `piccolo meta version`.
- Added example queries to the playground.

15.60 0.11.1

- Added `table_finder`, for use in `AppConfig`.
- Added support for concatenating strings using an update query.
- Added more tables to the playground, with more column types.
- Improved consistency between SQLite and Postgres with `UUID` columns, `Integer` columns, and `exists` queries.

15.61 0.11.0

Added `Numeric` and `Real` column types.

15.62 0.10.8

Fixing a bug where Postgres versions without a patch number couldn't be parsed.

15.63 0.10.7

Improving release script.

15.64 0.10.6

Sorting out packaging issue - old files were appearing in release.

15.65 0.10.5

Auto migrations can now run backwards.

15.66 0.10.4

Fixing some typos with `Table` imports. Showing a traceback when piccolo_conf can't be found by `engine_finder`.

15.67 0.10.3

Adding missing jinja templates to setup.py.

15.68 0.10.2

Fixing a bug when using `piccolo project new` in a new project.

15.69 0.10.1

Fixing bug with enum default values.

15.70 0.10.0

Using targ for the CLI. Refactored some core code into apps.

15.71 0.9.3

Suppressing exceptions when trying to find the Postgres version, to avoid an `ImportError` when importing *piccolo_conf.py*.

15.72 0.9.2

`.first()` bug fix.

15.73 0.9.1

Auto migration fixes, and `.first()` method now returns None if no match is found.

15.74 0.9.0

Added support for auto migrations.

15.75 0.8.3

Can use operators in update queries, and fixing 'new' migration command.

15.76 0.8.2

Fixing release issue.

15.77 0.8.1

Improved transaction support - can now use a context manager. Added `Secret`, `BigInt` and `SmallInt` column types. Foreign keys can now reference the parent table.

15.78 0.8.0

Fixing bug when joining across several tables. Can pass values directly into the `Table.update` method. Added `if_not_exists` option when creating a table.

15.79 0.7.7

Column sequencing matches the definition order.

15.80 0.7.6

Supporting *ON DELETE* and *ON UPDATE* for foreign keys. Recording reverse foreign key relationships.

15.81 0.7.5

Made `response_handler` async. Made it easier to rename columns.

15.82 0.7.4

Bug fixes and dependency updates.

15.83 0.7.3

Adding missing *__int__.py* file.

15.84 0.7.2

Changed migration import paths.

15.85 0.7.1

Added `remove_db_file` method to `SQLiteEngine` - makes testing easier.

15.86 0.7.0

Renamed `create` to `create_table`, and can register commands via *piccolo_conf*.

15.87 0.6.1

Adding missing *__init__.py* files.

15.88 0.6.0

Moved `BaseUser`. Migration refactor.

15.89 0.5.2

Moved drop table under `Alter` - to help prevent accidental drops.

15.90 0.5.1

Added `batch` support.

15.91 0.5.0

Refactored the `Table` Metaclass - much simpler now. Scoped more of the attributes on `Column` to avoid name clashes. Added `engine_finder` to make database configuration easier.

15.92 0.4.1

SQLite is now returning datetime objects for timestamp fields.

15.93 0.4.0

Refactored to improve code completion, along with bug fixes.

15.94 0.3.7

Allowing `Update` queries in SQLite

15.95 0.3.6

Falling back to *LIKE* instead of *ILIKE* for SQLite

15.96 0.3.5

Renamed `User` to `BaseUser`.

15.97 0.3.4

Added `ilike`.

15.98 0.3.3

Added value types to columns.

15.99 0.3.2

Default values infer the engine type.

15.100 0.3.1

Update click version.

15.101 0.3

Tweaked API to support more auto completion. Join support in where clause. Basic SQLite support - mostly for playground.

15.102 0.2

Using `QueryString` internally to represent queries, instead of raw strings, to harden against SQL injection.

15.103 0.1.2

Allowing joins across multiple tables.

15.104 0.1.1

Added playground.

CHAPTER 16

TLDR

Install Piccolo:

```
pip install piccolo
```

Experiment with queries:

```
piccolo playground run
```

Give me an ASGI web app!

```
piccolo asgi new
```

Index

Symbols

A

B

C

D

F

I

J

N

P

R

S

T

U

V