Piccolo

Release 0.64.0

Daniel Townsend

Jan 16, 2022

CONTENTS:

1 Getting Started 3

2 Query Types 9

3 Query Clauses 31

4 Schema 41

5 Projects and Apps 65

6 Engines 75

7 Migrations 81

8 Authentication 85

9 ASGI 89

10 Serialization 91

11 Testing 95

12 Features 99

13 Playground 101

14 Deployment 103

15 Ecosystem 105

16 Contributing 107

17 Changes 109

18 Help 147

19 TLDR 149

20 Videos 151

Index 153

Piccolo is a modern, async query builder and ORM for Python, with lots of batteries included.

CHAPTER

ONE

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 -m venv my_project
cd my_project
source bin/activate

# The important bit:
pip install piccolo

# Install Piccolo with PostgreSQL driver:
pip install 'piccolo[postgres]'

# Install Piccolo with SQLite driver:
pip install 'piccolo[sqlite]'

# Optional: orjson for improved JSON serialisation performance
pip install 'piccolo[orjson]'

# Optional: uvloop as the default event loop instead of asyncio
# If using Piccolo with Uvicorn, Uvicorn will set uvloop as the default
# event loop if installed
pip install 'piccolo[uvloop]'

# If you just want Piccolo with all of it's functionality, you might prefer
# to use this:
pip install 'piccolo[all]'
```

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 tested on most major Postgres versions (see the GitHub Actions 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())
```

Direct await

You can directly await a query if you prefer. For example:

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

By convention, we await the `run` method (`await Band.select().run()`), but you can use this shorter form if you prefer.

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

TWO

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 use 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.

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

The joins can go several layers deep.

```
>>> Concert.select(Concert.id, Concert.band_1.manager.name).run_sync()
[{'id': 1, 'band_1.manager.name': 'Guido'}]
```

all_columns

If you want all of the columns from a related table you can use `all_columns`, which is a useful shortcut which saves you from typing them all out:

```
>>> Band.select(Band.name, Band.manager.all_columns()).run_sync()
[
    {'name': 'Pythonistas', 'manager.id': 1, 'manager.name': 'Guido'},
    {'name': 'Rustaceans', 'manager.id': 2, 'manager.name': 'Graydon'}
]
```

In Piccolo < 0.41.0 you had to explicitly unpack `all_columns`. This is equivalent to the code above:

```
>>> Band.select(Band.name, *Band.manager.all_columns()).run_sync()
```

You can exclude some columns if you like:

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

Strings are supported too if you prefer:

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

You can also use `all_columns` on the root table, which saves you time if you have lots of columns. It works identically to related tables:

```
>>> Band.select(
>>>     Band.all_columns(exclude=[Band.id]),
>>>     Band.manager.all_columns(exclude=[Band.manager.id])
>>> ).run_sync()
[
    {'name': 'Pythonistas', 'popularity': 1000, 'manager.name': 'Guido'},
    {'name': 'Rustaceans', 'popularity': 500, 'manager.name': 'Graydon'}
]
```

Nested

You can also get the response as nested dictionaries, which can be very useful:

```
>>> Band.select(Band.name, Band.manager.all_columns()).output(nested=True).run_sync()
[
    {'name': 'Pythonistas', 'manager': {'id': 1, 'name': 'Guido'}},
    {'name': 'Rustaceans', 'manager': {'id': 2, 'manager.name': 'Graydon'}}
]
```

2.1.3 String syntax

You can specify the column names using a string if you prefer. 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 Aggregate functions

Count

Returns the number of rows which match the query:

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

Avg

Returns the average for a given column:

```
>>> from piccolo.query import Avg
>>> response = Band.select(Avg(Band.popularity)).first().run_sync()
>>> response["avg"]
750.0
```

Sum

Returns the sum for a given column:

```
>>> from piccolo.query import Sum
>>> response = Band.select(Sum(Band.popularity)).first().run_sync()
>>> response["sum"]
1500
```

Max

Returns the maximum for a given column:

```
>>> from piccolo.query import Max
>>> response = Band.select(Max(Band.popularity)).first().run_sync()
>>> response["max"]
1000
```

Min

Returns the minimum for a given column:

```
>>> from piccolo.query import Min
>>> response = Band.select(Min(Band.popularity)).first().run_sync()
>>> response["min"]
500
```

Additional features

You also can chain multiple different aggregate functions in one query:

```
>>> from piccolo.query import Avg, Sum
>>> response = Band.select(Avg(Band.popularity), Sum(Band.popularity)).first().run_sync()
>>> response
{"avg": 750.0, "sum": 1500}
```

And can use aliases for aggregate functions like this:

```
>>> from piccolo.query import Avg
>>> response = Band.select(Avg(Band.popularity, alias="popularity_avg")).first().run_
↪sync()
>>> response["popularity_avg"]
750.0

# Alternatively, you can use the `as_alias` method.
>>> response = Band.select(Avg(Band.popularity).as_alias("popularity_avg")).first().run_
↪sync()
>>> response["popularity_avg"]
750.0
```

2.1.5 Query clauses

batch

See *batch*.

columns

By default all columns are returned from the queried table.

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

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

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

# Or alternatively:
Band.select().columns(Band.name).run_sync()
```

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

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

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

first

See *first*.

group_by

See *group_by*.

limit

See *limit*.

offset

See *offset*.

distinct

See *distinct*.

order_by

See *order_by*.

output

See *output*.

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 a single row (or `None` if it doesn't exist):

```
>>> Band.objects().get(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()
```

This can also be done like this:

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

2.2.3 Updating objects

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

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

band.popularity = 100000

# This saves all values back to the database.
band.save().run_sync()

# Or specify specific columns to save:
band.save([Band.popularity]).run_sync()
```

2.2.4 Deleting objects

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

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

band.remove().run_sync()
```

2.2.5 Fetching related objects

get_related

If you have an object from a table with a `ForeignKey` column, and you want to fetch the related row as an object, you can do so using `get_related`.

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

manager = band.get_related(Band.manager).run_sync()
>>> manager
<Manager: 1>
>>> manager.name
'Guido'
```

Prefetching related objects

You can also prefetch the rows from related tables, and store them as child objects. To do this, pass `ForeignKey` columns into `objects`, which refer to the related rows you want to load.

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

>>> band.manager
<Manager: 1>
>>> band.manager.name
'Guido'
```

If you have a table containing lots of `ForeignKey` columns, and want to prefetch them all you can do so using `all_related`.

```
ticket = Ticket.objects(
    Ticket.concert.all_related()
).first().run_sync()

# Any intermediate objects will also be loaded:
>>> ticket.concert
```

(continues on next page)

(continued from previous page)

```
<Concert: 1>

>>> ticket.concert.band_1
<Band: 1>
>>> ticket.concert.band_2
<Band: 2>
```

You can manipulate these nested objects, and save the values back to the database, just as you would expect:

```
ticket.concert.band_1.name = 'Pythonistas 2'
ticket.concert.band_1.save().run_sync()
```

Instead of passing the `ForeignKey` columns into the `objects` method, you can use the `prefetch` clause if you prefer.

```
# These are equivalent:
ticket = Ticket.objects(
    Ticket.concert.all_related()
).first().run_sync()

ticket = Ticket.objects().prefetch(
    Ticket.concert.all_related()
).run_sync()
```

2.2.6 get_or_create

With `get_or_create` you can get an existing record matching the criteria, or create a new one with the `defaults` arguments:

```
band = Band.objects().get_or_create(
    Band.name == 'Pythonistas', defaults={Band.popularity: 100}
).run_sync()

# Or using string column names
band = Band.objects().get_or_create(
    Band.name == 'Pythonistas', defaults={'popularity': 100}
).run_sync()
```

You can find out if an existing row was found, or if a new row was created:

```
band = Band.objects.get_or_create(
    Band.name == 'Pythonistas'
).run_sync()
band._was_created  # True if it was created, otherwise False if it was already in the db
```

Complex where clauses are supported, but only within reason. For example:

```
# This works OK:
band = Band.objects().get_or_create(
    (Band.name == 'Pythonistas') & (Band.popularity == 1000),
).run_sync()
```

(continues on next page)

(continued from previous page)

```
# This is problematic, as it's unclear what the name should be if we
# need to create the row:
band = Band.objects().get_or_create(
    (Band.name == 'Pythonistas') | (Band.name == 'Rustaceans'),
    defaults={'popularity': 100}
).run_sync()
```

2.2.7 to_dict

If you need to convert an object into a dictionary, you can do so using the `to_dict` method.

```
band = Band.objects().first().run_sync()

>>> band.to_dict()
{'id': 1, 'name': 'Pythonistas', 'manager': 1, 'popularity': 1000}
```

If you only want a subset of the columns, or want to use aliases for some of the columns:

```
band = Band.objects().first().run_sync()

>>> band.to_dict(Band.id, Band.name.as_alias('title'))
{'id': 1, 'title': 'Pythonistas'}
```

2.2.8 Query clauses

batch

See *batch*.

limit

See *limit*.

offset

See *offset*.

first

See *first*.

order_by

See *order_by*.

output

See *output*.

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()
```

If you have several tables which you want to drop, you can use `drop_tables` instead. It will drop them in the correct order.

```
from piccolo.table import drop_tables

drop_tables(Band, Manager)
```

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 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()
[]
```

Also, you can create multiple tables at once.

This function will automatically sort tables based on their foreign keys so they're created in the right order:

```
>>> from piccolo.table import create_tables
>>> create_tables(Band, Manager, if_not_exists=True)
```

2.5 Delete

This deletes any matching rows from the table.

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

2.5.1 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()
[]
```

2.5.2 Query clauses

where

See *where*

2.6 Exists

This checks whether any rows exist which match the criteria.

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

2.6.1 Query clauses

where

See *where*.

2.7 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()
```

2.7.1 add

You can also compose it as follows:

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

2.8 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.

2.9 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.

2.9.1 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({
```

(continues on next page)

(continued from previous page)

```
    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.

2.9.2 Query clauses

where

See *where*.

2.10 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.

2.10.1 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()
```

2.10.2 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.

2.11 Comparisons

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

2.11.1 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

get

They are very similar, except Django raises an `ObjectDoesNotExist` exception if no match is found, whilst Piccolo returns `None`.

```
# Django
>>> Band.objects.get(name="Pythonistas")
<Band: 1>
>>> Band.objects.get(name="DOESN'T EXIST")  # ObjectDoesNotExist!

# Piccolo
>>> Band.objects().get(Band.name == 'Pythonistas').run_sync()
<Band: 1>
>>> Band.objects().get(Band.name == "DOESN'T EXIST").run_sync()
None
```

get_or_create

```
# Django
band, created = Band.objects.get_or_create(name="Pythonistas")
>>> band
<Band: 1>
>>> created
True

# Piccolo
>>> band = Band.objects().get_or_create(Band.name == 'Pythonistas').run_sync()
>>> band
<Band: 1>
>>> band._was_created
True
```

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().get(Band.name == 'Pythonistas').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().get(Band.name == 'Pythonistas').run_sync()
>>> band.remove().run_sync()
```

In bulk:

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

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

select_related

Django has an optimisation called `select_related` which reduces the number of SQL queries required when accessing related objects.

```
# Django
band = Band.objects.get(name='Pythonistas')
>>> band.manager  # This triggers another db query
<Manager: 1>

# Django, with select_related
band = Band.objects.select_related('manager').get(name='Pythonistas')
>>> band.manager  # Manager is pre-cached, so there's no extra db query
<Manager: 1>
```

Piccolo has something similar:

```
# Piccolo
band = Band.objects(Band.manager).get(name='Pythonistas')
>>> band.manager
<Manager: 1>
```

Database Settings

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

CHAPTER

THREE

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 distinct

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

- *Select*

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

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

3.3 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.

3.3.1 Count

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

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

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

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

Source

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

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.

```
Band.select(Band.name, Count()).group_by(Band.name).run()

# We can use an alias. These two are equivalent:

Band.select(
    Band.name, Count(alias="total")
).group_by(Band.name).run()

Band.select(
    Band.name,
    Count().as_alias("total")
).group_by(Band.name).run()
```

3.4 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()
```

3.5 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]
```

3.6 order_by

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

- *Select*
- *Objects*

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

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

To order by descending:

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

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

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

3.7 output

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

- *Select*
- *Objects*

3.7.1 Select queries only

as_json

To return the data as a JSON string:

```
>>> Band.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]'`.

as_list

If you're just querying a single column from a database table, you can use `as_list` to flatten the results into a single list.

```
>>> Band.select(Band.id).output(as_list=True).run_sync()
[1, 2]
```

nested

Output any data from related tables in nested dictionaries.

```
>>> Band.select(Band.name, Band.manager.name).first().output(nested=True).run_sync()
{'name': 'Pythonistas', 'manager': {'name': 'Guido'}}
```

3.7.2 Select and Objects queries

load_json

If querying JSON or JSONB columns, you can tell Piccolo to deserialise the JSON values automatically.

```
>>> RecordingStudio.select().output(load_json=True).run_sync()
[{'id': 1, 'name': 'Abbey Road', 'facilities': {'restaurant': True, 'mixing_desk': True}}
↪]

>>> studio = RecordingStudio.objects().first().output(load_json=True).run_sync()
>>> studio.facilities
{'restaurant': True, 'mixing_desk': True}
```

3.8 where

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

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

It allows powerful filtering of your data.

3.8.1 Equal / Not Equal

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

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

Hint: With `Boolean` columns, some linters will complain if you write `SomeTable.some_column == True` (because it's more Pythonic to do `is True`). To work around this, you can do `SomeTable.some_column.eq(True)`. Likewise, with `!=` you can use `SomeTable.some_column.ne(True)`

3.8.2 Greater than / less than

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

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

3.8.3 like / ilike

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

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

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

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

Band.select().where(
    Band.name.like('Pythonistas')  # Matches the entire string
).run_sync()
```

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

3.8.4 not_like

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

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

3.8.5 is_in / not_in

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

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

3.8.6 is_null / is_not_null

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

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

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

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

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

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

3.8.7 Complex queries - and / or

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

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

Band.select().where(
    (Band.popularity >= 100) | (Band.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.

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

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

Also, multiple arguments inside `where` clause is equivalent to an AND.

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

Band.select().where(
    Band.popularity >= 100, Band.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

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

3.8.8 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

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

3.9 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())
```

3.10 freeze

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

3.10.1 Source

`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

FOUR

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*.

Hint: If you're using an existing database, see Piccolo's *auto schema generation command*, which will save you some time.

4.1.1 Primary Key

Piccolo tables are automatically given a primary key column called `id`, which is an auto incrementing integer.

There is currently experimental support for specifying a custom primary key column. For example:

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


class Band(Table):
    id = UUID(primary_key=True)
    name = Varchar(length=100)
```

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_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*, *db_column_name: Optional[str] = None*, *secret: bool = False*, ***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_key** – If set, the column is used as a primary 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.
- **choices** – An optional Enum - when specified, other tools such as Piccolo Admin will render the available options in the GUI.
- **db_column_name** – If specified, you can override the name used for the column in the database. The main reason for this is when using a legacy database, with a problematic column name (for example `'class'`, which is a reserved Python keyword). Here's an example:

```
class MyTable(Table):
    class_ = Varchar(db_column_name="class")

>>> MyTable.select(MyTable.class_).run_sync()
[{'id': 1, 'class': 'test'}]
```

  This is an advanced feature which you should only need in niche situations.

- **secret** – If `secret=True` is specified, it allows a user to automatically omit any fields when doing a select query, to help prevent inadvertent leakage of sensitive data.

```
class Band(Table):
    name = Varchar()
    net_worth = Integer(secret=True)

>>> Property.select(exclude_secrets=True).run_sync()
[{'name': 'Pythonistas'}]
```

4.2.2 Bytea

class piccolo.columns.column_types.**Bytea**(*default: Optional[Union[bytes, bytearray, enum.Enum, Callable[[], bytes], Callable[[], bytearray]]] = 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: Optional[Union[bool, enum.Enum, Callable[[], bool]]] = 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.Any = None*, *null: bool = True*, *on_delete: OnDelete = OnDelete.cascade*, *on_update: OnUpdate = OnUpdate.cascade*, *target_column: t.Union[str,* Column*, None] = None*, ***kwargs*)

Used to reference another table. Uses the same type as the primary key column on the table it references.

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

You also use it to perform joins:

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

To retrieve all of the columns in the related table:

```
>>> await Band.select(Band.name, *Band.manager.all_columns()).first().run()
{'name': 'Pythonistas', 'manager.id': 1, '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

```

(continues on next page)

(continued from previous page)

```
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(
        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(
```

(continues on next page)

(continued from previous page)

```
        references=Manager,
        on_update=OnUpdate.cascade
    )
```

- **target_column** – By default the `ForeignKey` references the primary key column on the related table. You can specify an alternative column (it must have a unique constraint on it though). For example:

```
# Passing in a column reference:
ForeignKey(references=Manager, target_column=Manager.passport_number)

# Or just the column name:
ForeignKey(references=Manager, target_column='passport_number')
```

4.2.5 Number

BigInt

class piccolo.columns.column_types.**BigInt**(*default: Optional[Union[int, enum.Enum, Callable[[], int]]] = 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}
```

BigSerial

class piccolo.columns.column_types.**BigSerial**(*null: bool = False*, *primary_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*, *db_column_name: Optional[str] = None*, *secret: bool = False*, ***kwargs*)

An alias to a large autoincrementing integer column in Postgres.

Double Precision

class piccolo.columns.column_types.**DoublePrecision**(*default: Optional[Union[float, enum.Enum, Callable[[], float]]] = 0.0*, ***kwargs*)

The same as `Real`, except the numbers are stored with greater precision.

Integer

class piccolo.columns.column_types.**Integer**(*default: Optional[Union[int, enum.Enum, Callable[[], int]]] = 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: Optional[Union[decimal.Decimal, enum.Enum, Callable[[], decimal.Decimal]]] = 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: Optional[Union[float, enum.Enum, Callable[[], float]]] = 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`.

Serial

class piccolo.columns.column_types.**Serial**(*null: bool = False*, *primary_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*, *db_column_name: Optional[str] = None*, *secret: bool = False*, ***kwargs*)

An alias to an autoincrementing integer column in Postgres.

SmallInt

class piccolo.columns.column_types.**SmallInt**(*default: Optional[Union[int, enum.Enum, Callable[[], int]]] = 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()
```

(continues on next page)

(continued from previous page)

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

4.2.6 UUID

class piccolo.columns.column_types.**UUID**(*default: Optional[Union[piccolo.columns.defaults.uuid.UUID4, uuid.UUID, str, enum.Enum]] = UUID4()*, ***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
>>> 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**(**args*, ***kwargs*)

This is just an alias to `Varchar(secret=True)`. It's here for backwards compatibility.

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()
```

(continues on next page)

(continued from previous page)

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

Varchar

class piccolo.columns.column_types.**Varchar**(*length: int = 255*, *default: Optional[Union[str, enum.Enum, Callable[[], str]]] = ''*, ***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] = DateNow()*, ***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
```

(continues on next page)

(continued from previous page)

```
>>> 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] = IntervalCustom(weeks=0, days=0, hours=0, minutes=0, seconds=0, milliseconds=0, microseconds=0)*, ***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] = TimeNow()*, ***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.TimestampCustom, piccolo.columns.defaults.timestamp.TimestampNow, piccolo.columns.defaults.timestamp.TimestampOffset, enum.Enum, None, datetime.datetime, piccolo.columns.defaults.timestamp.DatetimeDefault] = TimestampNow()*, ***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.TimestamptzCustom, piccolo.columns.defaults.timestamptz.TimestamptzNow, piccolo.columns.defaults.timestamptz.TimestamptzOffset, enum.Enum, None, datetime.datetime] = TimestamptzNow()*, ***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()
```

(continues on next page)

(continued from previous page)

```
# Query
>>> Concert.select(Concert.starts).run_sync()
{
    '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: Optional[Union[str, List, Dict, Callable[[], Union[str, List, Dict]]]] = '{}'*, ***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: Optional[Union[str, List, Dict, Callable[[], Union[str, List, Dict]]]] = '{}'*, ***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"}'),
```

(continues on next page)

(continued from previous page)

```
    Booking(data='{"name": "Bob"}')
).run_sync()

# 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: Optional[Union[List, enum.Enum, Callable[[], List]]] = <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.

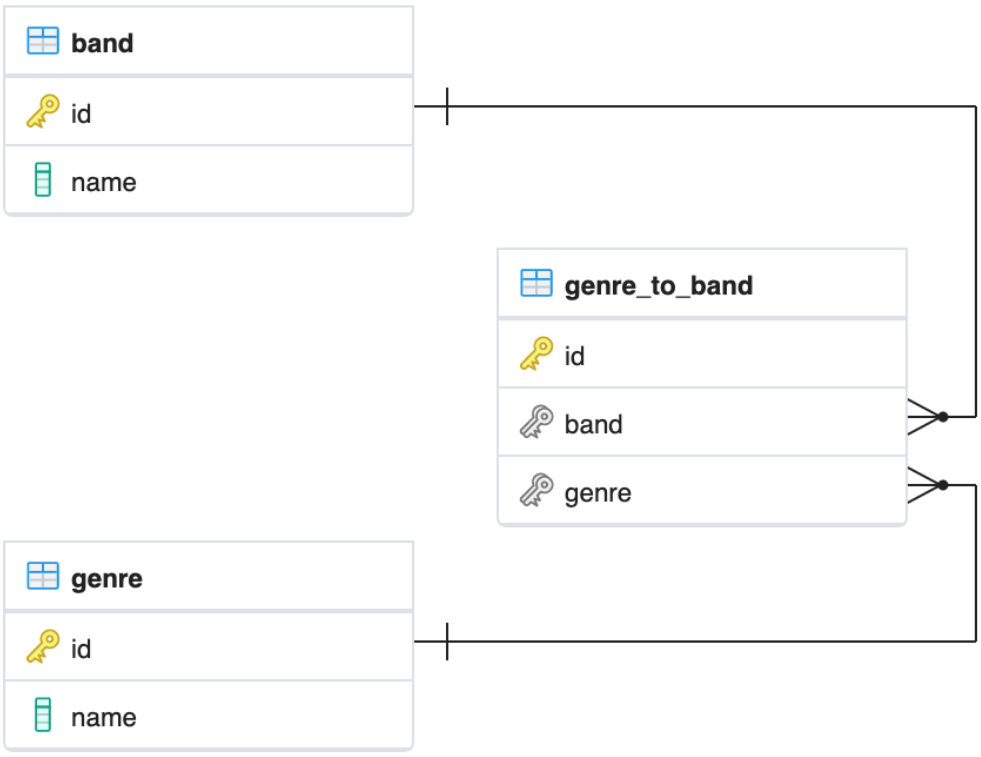
```
>>> Ticket.select().where(Ticket.seat_numbers.all(510)).run_sync()
```

4.3 M2M

Sometimes in database design you need many-to-many (M2M) relationships.

For example, we might have our `Band` table, and want to describe which genres of music each band belongs to (e.g. rock and electronic). As each band can have multiple genres, a `ForeignKey` on the `Band` table won't suffice. Our options are using an `Array` / `JSON` / `JSONB` column, or using an `M2M` relationship.

Postgres and SQLite don't natively support `M2M` relationships - we create them using a joining table which has foreign keys to each of the related tables (in our example, `Genre` and `Band`).



We create it in Piccolo like this:

```
from piccolo.columns.column_types import (
    ForeignKey,
    LazyTableReference,
    Varchar
)
from piccolo.columns.m2m import M2M
from piccolo.table import Table


class Band(Table):
    name = Varchar()
    genres = M2M(LazyTableReference("GenreToBand", module_path=__name__))


class Genre(Table):
    name = Varchar()
    bands = M2M(LazyTableReference("GenreToBand", module_path=__name__))


# This is our joining table:
class GenreToBand(Table):
    band = ForeignKey(Band)
    genre = ForeignKey(Genre)
```

Note: We use `LazyTableReference` because when Python evaluates `Band` and `Genre`, the `GenreToBand` class doesn't exist yet.

By using `M2M` it unlocks some powerful and convenient features.

4.3.1 Select queries

If we want to select each band, along with a list of genres that they belong to, we can do this:

```
>>> await Band.select(Band.name, Band.genres(Genre.name, as_list=True))
[
    {"name": "Pythonistas", "genres": ["Rock", "Folk"]},
    {"name": "Rustaceans", "genres": ["Folk"]},
    {"name": "C-Sharps", "genres": ["Rock", "Classical"]},
]
```

You can request whichever column you like from the related table:

```
>>> await Band.select(Band.name, Band.genres(Genre.id, as_list=True))
[
    {"name": "Pythonistas", "genres": [1, 2]},
    {"name": "Rustaceans", "genres": [2]},
    {"name": "C-Sharps", "genres": [1, 3]},
]
```

You can also request multiple columns from the related table:

```
>>> await Band.select(Band.name, Band.genres(Genre.id, Genre.name))
[
    {
        'name': 'Pythonistas',
        'genres': [
            {'id': 1, 'name': 'Rock'},
            {'id': 2, 'name': 'Folk'}
        ]
    },
    ...
]
```

If you omit the columns argument, then all of the columns are returned.

```
>>> await Band.select(Band.name, Band.genres())
[
    {
        'name': 'Pythonistas',
        'genres': [
            {'id': 1, 'name': 'Rock'},
            {'id': 2, 'name': 'Folk'}
        ]
    },
    ...
]
```

As we defined `M2M` on the `Genre` table too, we can get each band in a given genre:

```
>>> await Genre.select(Genre.name, Genre.bands(Band.name, as_list=True))
[
    {"name": "Rock", "bands": ["Pythonistas", "C-Sharps"]},
    {"name": "Folk", "bands": ["Pythonistas", "Rustaceans"]},
    {"name": "Classical", "bands": ["C-Sharps"]},
]
```

4.3.2 Objects queries

Piccolo makes it easy working with objects and `M2M` relationship.

add_m2m

`Table.add_m2m`(**rows: piccolo.table.Table, m2m: piccolo.columns.m2m.M2M, extra_column_values: Dict[Union[piccolo.columns.base.Column, str], Any] = {}*) → piccolo.columns.m2m.M2MAddRelated

Save the row if it doesn't already exist in the database, and insert an entry into the joining table.

```
>>> band = await Band.objects().get(Band.name == "Pythonistas")
>>> await band.add_m2m(
>>>     Genre(name="Punk rock"),
>>>     m2m=Band.genres
```

(continues on next page)

(continued from previous page)

```
>>> )
[{'id': 1}]
```

Parameters **extra_column_values** – If the joining table has additional columns besides the two required foreign keys, you can specify the values for those additional columns. For example, if this is our joining table:

```
class GenreToBand(Table):
    band = ForeignKey(Band)
    genre = ForeignKey(Genre)
    reason = Text()
```

We can provide the `reason` value:

```
await band.add_m2m(
    Genre(name="Punk rock"),
    m2m=Band.genres,
    extra_column_values={
        "reason": "Their second album was very punk."
    }
)
```

get_m2m

`Table.`**`get_m2m`**(*m2m: piccolo.columns.m2m.M2M*) → piccolo.columns.m2m.M2MGetRelated

Get all matching rows via the join table.

```
>>> band = await Band.objects().get(Band.name == "Pythonistas")
>>> await band.get_m2m(Band.genres)
[<Genre: 1>, <Genre: 2>]
```

remove_m2m

`Table.`**`remove_m2m`**(**rows: piccolo.table.Table*, *m2m: piccolo.columns.m2m.M2M*) → piccolo.columns.m2m.M2MRemoveRelated

Remove the rows from the joining table.

```
>>> band = await Band.objects().get(Band.name == "Pythonistas")
>>> genre = await Genre.objects().get(Genre.name == "Rock")
>>> await band.remove_m2m(
>>>     genre,
>>>     m2m=Band.genres
>>> )
```

Hint: All of these methods can be run synchronously as well - for example, `band.get_m2m(Band.genres).run_sync()`.

4.4 Advanced

4.4.1 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.

4.4.2 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)
```

4.4.3 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()
```

4.4.4 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

4.4.5 Reflection

This is a very advanced feature, which is only required for specialist use cases. Currently, just Postgres is supported.

Instead of writing your `Table` definitions in a `tables.py` file, Piccolo can dynamically create them at run time, by inspecting the database. These `Table` classes are then stored in memory, using a singleton object called `TableStorage`.

Some example use cases:

- You have a very dynamic database, where new tables are being created constantly, so updating a `tables.py` is impractical.
- You use Piccolo on the command line to explore databases.

Full reflection

Here's an example, where we reflect the entire schema:

```
from piccolo.table_reflection import TableStorage

storage = TableStorage()
await storage.reflect(schema_name="music")
```

`Table` objects are accessible from `TableStorage.tables`:

```
>>> storage.tables
{"music.Band": <class 'Band'>, ... }

>>> Band = storage.tables["music.Band"]
```

Then you can use them like your normal `Table` classes:

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

Partial reflection

Full schema reflection can be a heavy process based on the size of your schema. You can use `include`, `exclude` and `keep_existing` parameters of the `reflect` method to limit the overhead dramatically.

Only reflect the needed table(s):

```
from piccolo.table_reflection import TableStorage

storage = TableStorage()
await storage.reflect(schema_name="music", include=['band', ...])
```

Exclude table(s):

```
await storage.reflect(schema_name="music", exclude=['band', ...])
```

If you set `keep_existing=True`, only new tables on the database will be reflected and the existing tables in `TableStorage` will be left intact.

```
await storage.reflect(schema_name="music", keep_existing=True)
```

get_table

`TableStorage` has a helper method named `get_table`. If the table is already present in the `TableStorage`, this will return it and if the table is not present, it will be reflected and returned.

```
Band = storage.get_table(tablename='band')
```

Hint: Reflection will automatically create `Table` classes for referenced tables too. For example, if `Table1` references `Table2`, then `Table2` will automatically be added to `TableStorage`.

CHAPTER

FIVE

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, use the following command:

```
piccolo project new
```

The file serves two important purposes:

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

Location

By convention, the `piccolo_conf.py` file should be at the root of your project:

```
my_project/
    piccolo_conf.py
    my_app/
        piccolo_app.py
```

This means that when you use the `piccolo` CLI from the `my_project` folder it can import `piccolo_conf.py`.

If you prefer to keep `piccolo_conf.py` in a different location, or to give it a different name, you can do so using the `PICCOLO_CONF` environment variable (see *PICCOLO_CONF*). For example:

```
my_project/
    conf/
        piccolo_conf_local.py
    my_app/
        piccolo_app.py
```

```
export PICCOLO_CONF=conf.piccolo_conf_local
```

5.1.2 Example

Here's an example:

```
from piccolo.engine.postgres import PostgresEngine


from piccolo.conf.apps import AppRegistry


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__))
```

(continues on next page)

(continued from previous page)

```
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*).

Source

piccolo.conf.apps.**table_finder**(*modules: Sequence[str]*, *include_tags: Sequence[str] = ['__all__']*, *exclude_tags: Sequence[str] = []*, *exclude_imported: bool = False*) → 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`.
- **exclude_imported** – If `True`, only `Table` subclasses defined within the module are used. Any `Table` subclasses imported by that module from other modules are ignored. For example:

```
from piccolo.table import Table
from piccolo.column import Varchar, ForeignKey
from piccolo.apps.user.tables import BaseUser # excluded

class Task(Table): # included
    title = Varchar()
    creator = ForeignKey(BaseUser)
```

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("hello,", 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
hello, bob
```

If the code contains an error to see more details in the output add a `--trace` flag to the command line.

```
>>> piccolo my_app say_hello bob --trace
```

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.

Hint: To find out more about each of these commands you can use the `--help` flag on the command line. For example `piccolo app new --help`.

---

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
```

---

fixtures

Fixtures are used when you want to seed your database with essential data (for example, country names).

Once you have created a fixture, it can be used by your colleagues when setting up an application on their local machines, or when deploying to a new environment.

Databases such as Postgres have inbuilt ways of dumping and restoring data (via `pg_dump` and `pg_restore`). Some reasons to use the fixtures app instead:

- When you want the data to be loadable in a range of database versions.
- Fixtures are stored in JSON, which are a bit friendlier for source control.

To dump the data into a new fixture file:

```
piccolo fixtures dump > fixtures.json
```

By default, the fixture contains data from all apps and tables. You can specify a subset of apps and tables instead, for example:

```
piccolo fixtures dump --apps=blog --tables=Post > fixtures.json

# Or for multiple apps / tables
piccolo fixtures dump --apps=blog,shop --tables=Post,Product > fixtures.json
```

To load the fixture:

```
piccolo fixtures load fixtures.json
```

---

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
```

---

schema

generate

Lets you auto generate Piccolo `Table` classes from an existing database. Make sure the credentials in `piccolo_conf.py` are for the database you're interested in, then run the following:

```
piccolo schema generate > tables.py
```

Warning: This feature is still a work in progress. However, even in it's current form it will save you a lot of time. Make sure you check the generated code to make sure it's correct.

graph

A basic schema visualisation tool. It prints out the contents of a GraphViz dot file representing your schema.

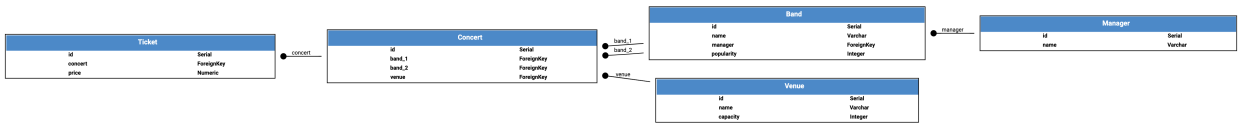
```
piccolo schema graph
```

You can pipe the output to your clipboard (`piccolo schema graph | pbcopy` on a Mac), then paste it into a website like this to turn it into an image file.

Or if you have Graphviz installed on your machine, you can do this to create an image file:

```
piccolo schema graph | dot -Tpdf -o graph.pdf
```

Here's an example of a generated image:



---

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).

---

tester

Launches pytest , which runs your unit test suite. The advantage of using this rather than running `pytest` directly, is the `PICCOLO_CONF` environment variable will automatically be set before the testing starts, and will be restored to it's initial value once the tests finish.

```
piccolo tester run
```

Setting the *PICCOLO_CONF* environment variable means your code will use the database engine specified in that file for the duration of the testing.

By default `piccolo tester run` sets `PICCOLO_CONF` to `'piccolo_conf_test'`, meaning that a file called `piccolo_conf_test.py` will be imported.

Within the `piccolo_conf_test.py` file, override the database settings, so it uses a test database:

```
from piccolo_conf import *

DB = PostgresEngine(
    config={
        "database": "my_app_test"
    }
)
```

If you prefer, you can set a custom `PICCOLO_CONF` value:

```
piccolo tester run --piccolo_conf=my_custom_piccolo_conf
```

You can also pass arguments to pytest:

```
piccolo tester run --pytest_args="-s foo"
```

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

SIX

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.py` 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 of 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.

Hint: Piccolo has a builtin command which will do this for you - automatically setting `PICCOLO_CONF` for the duration of your tests. See *tester*.

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


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

It's also 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.

If the `piccolo_conf.py` file is located in a sub-module (rather than the root of your project) you can specify the path like this:

```
export PICCOLO_CONF=sub_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=3*, *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

SEVEN

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/
    2021-08-06T16-22-51-415781.py
```

The contents of an empty migration file looks like this:

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


ID = '2021-08-06T16:22:51:415781'
VERSION = "0.29.0" # The version of Piccolo used to create it
DESCRIPTION = "Optional description"


async def forwards():
    manager = MigrationManager(migration_id=ID, app_name="my_app",␣
↪description=DESCRIPTION)

    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 = '2021-08-06T16:22:51:415781'
VERSION = "0.29.0" # The version of Piccolo used to create it
DESCRIPTION = "Optional description"


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.

7.1.3 Migration descriptions

To make the migrations more memorable, you can give them a description. Inside the migration file, you can set a `DESCRIPTION` global variable manually, or can specify it when creating the migration:

```
piccolo migrations new my_app --auto --desc="Adding name column"
```

The Piccolo CLI will then use this description when listing migrations, to make them easier to identify.

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

EIGHT

AUTHENTICATION

Piccolo ships with 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.

create

Creates a new user. It presents an interactive prompt, asking for the username, password etc.

```
piccolo user create
```

If you'd prefer to create a user without the interactive prompt (perhaps in a script), you can pass all of the arguments in as follows:

```
piccolo user create --username=bob --password=bob123 --email=foo@bar.com  --is_admin=t --
↪is_superuser=t --is_active=t
```

If you choose this approach then be careful, as the password will be in the shell's history.

change_password

Change a user's password.

```
piccolo user change_password
```

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.

Limits

The maximum password length allowed is 128 characters. This should be sufficiently long for most use cases.

8.3 Web app integration

Our sister project, Piccolo API, contains powerful endpoints and middleware for integrating session auth and token auth into your ASGI web application, using `BaseUser`.

CHAPTER

NINE

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

TEN

SERIALIZATION

Piccolo uses Pydantic internally to serialize and deserialize data.

10.1 create_pydantic_model

Using `create_pydantic_model` we can easily create a Pydantic model from a Piccolo `Table`.

Using this example schema:

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

class Manager(Table):
    name = Varchar()

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

Creating a Pydantic model is as simple as:

```
from piccolo.utils.pydantic import create_pydantic_model

BandModel = create_pydantic_model(Band)
```

We can then create model instances from data we fetch from the database:

```
# If using objects:
model = BandModel(
    **Band.objects().get(Band.name == 'Pythonistas').run_sync().to_dict()
)

# If using select:
model = BandModel(
    **Band.select().where(Band.name == 'Pythonistas').first().run_sync()
)

>>> model.name
'Pythonistas'
```

You have several options for configuring the model, as shown below.

10.1.1 include_columns / exclude_columns

If we want to exclude the `popularity` column from the `Band` table:

```
BandModel = create_pydantic_model(Band, exclude_columns=(Band.popularity,))
```

Conversely, if you only wanted the `popularity` column:

```
BandModel = create_pydantic_model(Band, include_columns=(Band.popularity,))
```

10.1.2 nested

Another great feature is `nested=True`. For each `ForeignKey` in the Piccolo `Table`, the Pydantic model will contain a sub model for the related table.

For example:

```
BandModel = create_pydantic_model(Band, nested=True)
```

If we were to write `BandModel` by hand instead, it would look like this:

```
from pydantic import BaseModel

class ManagerModel(BaseModel):
    name: str

class BandModel(BaseModel):
    name: str
    manager: ManagerModel
    popularity: int
```

But with `nested=True` we can achieve this with one line of code.

To populate a nested Pydantic model with data from the database:

```
# If using objects:
model = BandModel(
    **Band.objects(Band.manager).get(Band.name == 'Pythonistas').run_sync().to_dict()
)

# If using select:
model = BandModel(
    **Band.select(
        Band.all_columns(),
        Band.manager.all_columns()
    ).where(
        Band.name == 'Pythonistas'
    ).first().output(
        nested=True
    ).run_sync()
)
```

(continues on next page)

(continued from previous page)

```
>>> model.manager.name
'Guido'
```

10.1.3 include_default_columns

Sometimes you'll want to include the Piccolo `Table`'s primary key column in the generated Pydantic model. For example, in a `GET` endpoint, we usually want to include the `id` in the response:

```
// GET /api/bands/1/
// Response:
{"id": 1, "name": "Pythonistas", "popularity": 1000}
```

Other times, you won't want the Pydantic model to include the primary key column. For example, in a `POST` endpoint, when using a Pydantic model to serialise the payload, we don't expect the user to pass in an `id` value:

```
// POST /api/bands/
// Payload:
{"name": "Pythonistas", "popularity": 1000}
```

By default the primary key column isn't included - you can add it using:

```
BandModel = create_pydantic_model(Band, include_default_columns=True)
```

10.1.4 Source

piccolo.utils.pydantic.**create_pydantic_model**(*table: Type[piccolo.table.Table]*, *nested: Union[bool, Tuple[piccolo.columns.column_types.ForeignKey, ...]] = False*, *exclude_columns: Tuple[piccolo.columns.base.Column, ...] = ()*, *include_columns: Tuple[piccolo.columns.base.Column, ...] = ()*, *include_default_columns: bool = False*, *include_readable: bool = False*, *all_optional: bool = False*, *model_name: Optional[str] = None*, *deserialize_json: bool = False*, *recursion_depth: int = 0*, *max_recursion_depth: int = 5*, ***schema_extra_kwargs*) → Type[pydantic.main.BaseModel]

Create a Pydantic model representing a table.

Parameters

- **table** – The Piccolo `Table` you want to create a Pydantic serialiser model for.
- **nested** – Whether `ForeignKey` columns are converted to nested Pydantic models. If `False`, none are converted. If `True`, they all are converted. If a tuple of `ForeignKey` columns is passed in, then only those are converted.
- **exclude_columns** – A tuple of `Column` instances that should be excluded from the Pydantic model. Only specify `include_columns` or `exclude_columns`.
- **include_columns** – A tuple of `Column` instances that should be included in the Pydantic model. Only specify `include_columns` or `exclude_columns`.

- **include_default_columns** – Whether to include columns like `id` in the serialiser. You will typically include these columns in GET requests, but don't require them in POST requests.
- **include_readable** – Whether to include 'readable' columns, which give a string representation of a foreign key.
- **all_optional** – If True, all fields are optional. Useful for filters etc.
- **model_name** – By default, the classname of the Piccolo `Table` will be used, but you can override it if you want multiple Pydantic models based off the same Piccolo table.
- **deserialize_json** – By default, the values of any Piccolo `JSON` or `JSONB` columns are returned as strings. By setting this parameter to True, they will be returned as objects.
- **schema_extra_kwargs** – This can be used to add additional fields to the schema. This is very useful when using Pydantic's JSON Schema features. For example:

```
>>> my_model = create_pydantic_model(Band, my_extra_field="Hello")
>>> my_model.schema()
{..., "my_extra_field": "Hello"}
```

Recursion_depth Not to be set by the user - used internally to track recursion.

Max_recursion_depth If using nested models, this specifies the max amount of recursion.

Returns A Pydantic model.

Hint: A good place to see `create_pydantic_model` in action is PiccoloCRUD, as it uses `create_pydantic_model` extensively to create Pydantic models from Piccolo tables.

10.2 FastAPI template

Piccolo's FastAPI template uses `create_pydantic_model` to create serializers.

To create a new FastAPI app using Piccolo, simply use:

```
piccolo asgi new
```

See the *ASGI docs* for more details.

CHAPTER

ELEVEN

TESTING

Piccolo provides a few tools to make testing easier and decrease manual work.

11.1 Model Builder

When writing unit tests, it's usually required to have some data seeded into the database. You can build and save the records manually or use `ModelBuilder` to generate random records for you.

This way you can randomize the fields you don't care about and specify important fields explicitly and reduce the amount of manual work required. `ModelBuilder` currently supports all Piccolo column types and features.

Let's say we have the following schema:

```
from piccolo.columns import ForeignKey, Varchar

class Manager(Table):
    name = Varchar(length=50)

class Band(Table):
    name = Varchar(length=50)
    manager = ForeignKey(Manager, null=True)
```

You can build a random `Band` which will also build and save a random `Manager`:

```
from piccolo.testing.model_builder import ModelBuilder

band = await ModelBuilder.build(Band)  # Band instance with random values persisted
```

Note: `ModelBuilder.build(Band)` persists the record into the database by default.

You can also run it synchronously if you prefer:

```
manager = ModelBuilder.build_sync(Manager)
```

To specify any attribute, pass the `defaults` dictionary to the `build` method:

```
manager = ModelBuilder.build(Manager)

# Using table columns
band = await ModelBuilder.build(Band, defaults={Band.name: "Guido", Band.manager:␣
↪manager})

# Or using strings as keys
band = await ModelBuilder.build(Band, defaults={"name": "Guido", "manager": manager})
```

To build objects without persisting them into the database:

```
band = await ModelBuilder.build(Band, persist=False)
```

To build object with minimal attributes, leaving nullable fields empty:

```
band = await ModelBuilder.build(Band, minimal=True)  # Leaves manager empty
```

11.2 Test runner

This runs your unit tests using pytest. See the *tester app*.

11.3 Creating the test schema

When running your unit tests, you usually start with a blank test database, create the tables, and then install test data.

To create the tables, there are a few different approaches you can take. Here we use `create_tables` and `drop_tables`:

```
from unittest import TestCase

from piccolo.table import create_tables, drop_tables
from piccolo.conf.apps import Finder

TABLES = Finder().get_table_classes()

class TestApp(TestCase):
    def setUp(self):
        create_tables(*TABLES)

    def tearDown(self):
        drop_tables(*TABLES)

    def test_app(self):
        # Do some testing ...
        pass
```

Alternatively, you can run the migrations to setup the schema if you prefer:

```
import asyncio
from unittest import TestCase

from piccolo.apps.migrations.commands.backwards import run_backwards
from piccolo.apps.migrations.commands.forwards import run_forwards

class TestApp(TestCase):
    def setUp(self):
        asyncio.run(run_forwards("all"))

    def tearDown(self):
        asyncio.run(run_backwards("all", auto_agree=True))

    def test_app(self):
        # Do some testing ...
        pass
```

CHAPTER

TWELVE

FEATURES

12.1 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.

12.2 Supported Databases

12.2.1 Postgres

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

12.2.2 SQLite

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

12.3 Security

12.3.1 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.

12.4 Syntax

12.4.1 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.

12.4.2 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

THIRTEEN

PLAYGROUND

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

13.1 Advanced Playground Usage

13.1.1 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

FOURTEEN

DEPLOYMENT

14.1 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

FIFTEEN

ECOSYSTEM

15.1 Piccolo API

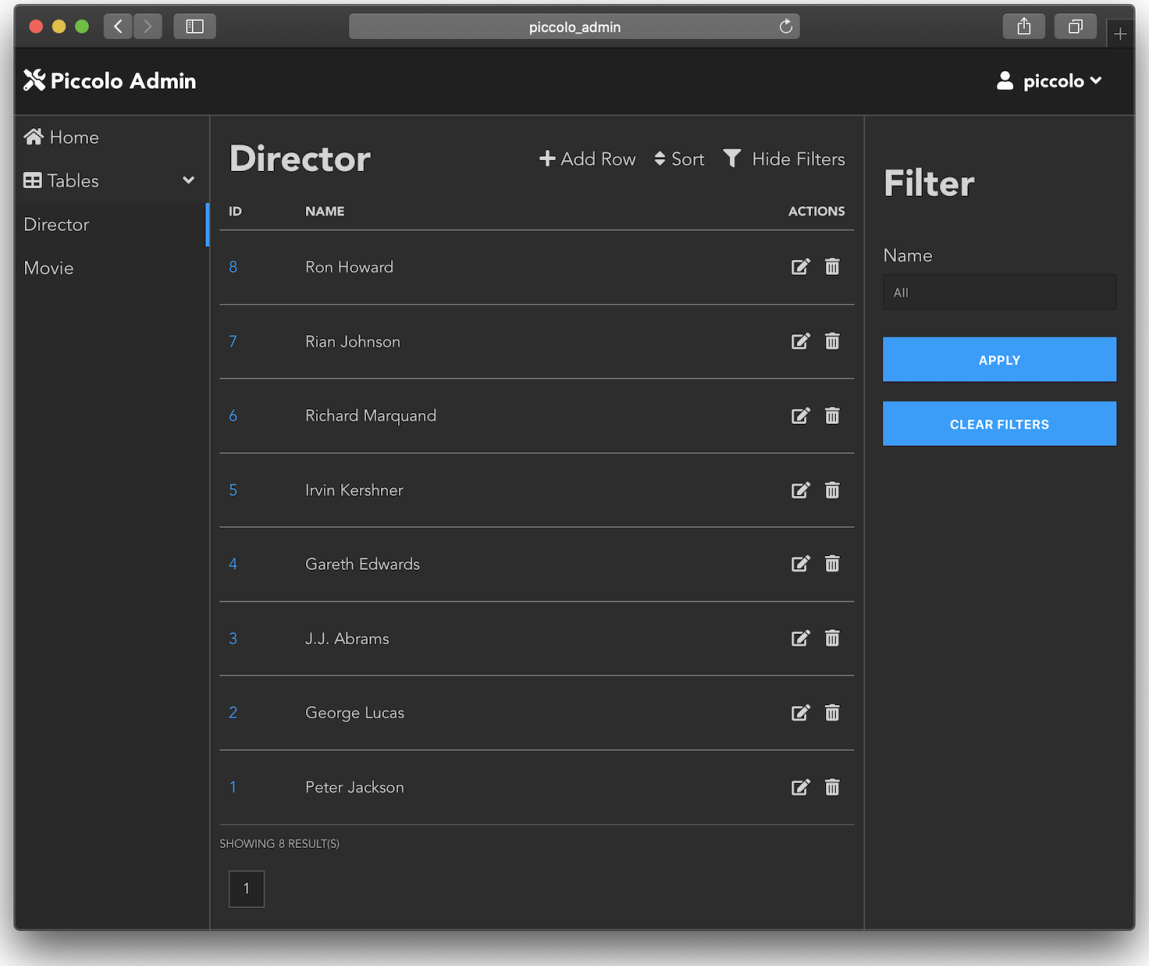
Provides some handy utilities for creating an API around your Piccolo tables. Examples include:

- Easily creating CRUD endpoints for ASGI apps, based on Piccolo tables.
- Automatically creating Pydantic models from your Piccolo tables.
- Great FastAPI integration.
- Authentication and rate limiting.

See the docs for more information.

15.2 Piccolo Admin

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



It's a modern UI built with Vue JS, which supports powerful data filtering, and CSV exports. It's the crown jewel in the Piccolo ecosystem!

15.3 Piccolo Examples

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

CHAPTER

SIXTEEN

CONTRIBUTING

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

16.1 Get the tests running

- Create a new virtualenv
- Clone the Git repo
- `cd piccolo`
- Install default dependencies: `pip install -r requirements/requirements.txt`
- Install development dependencies: `pip install -r requirements/dev-requirements.txt`
- Install test dependencies: `pip install -r requirements/test-requirements.txt`
- Setup Postgres
- Run the automated code linting/formatting tools: `./scripts/lint.sh`
- Run the test suite with Postgres: `./scripts/test-postgres.sh`
- Run the test suite with Sqlite: `./scripts/test-sqlite.sh`

16.2 Contributing to the docs

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

- Install the requirements: `pip install -r requirements/doc-requirements.txt`
- `cd docs`
- 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.

16.3 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

SEVENTEEN

CHANGES

17.1 0.64.0

Added initial support for `ForeignKey` columns referencing non-primary key columns. For example:

```
class Manager(Table):
    name = Varchar()
    email = Varchar(unique=True)

class Band(Table):
    manager = ForeignKey(Manager, target_column=Manager.email)
```

Thanks to @theelderbeever for suggesting this feature, and with help testing.

17.2 0.63.1

Fixed an issue with the `value_type` of `ForeignKey` columns when referencing a table with a custom primary key column (such as a `UUID`).

17.3 0.63.0

Added an `exclude_imported` option to `table_finder`.

```
APP_CONFIG = AppConfig(
    table_classes=table_finder(['music.tables'], exclude_imported=True)
)
```

It's useful when we want to import `Table` subclasses defined within a module itself, but not imported ones:

```
# tables.py
from piccolo.apps.user.tables import BaseUser # excluded
from piccolo.columns.column_types import ForeignKey, Varchar
from piccolo.table import Table
```

(continues on next page)

(continued from previous page)

```
class Musician(Table): # included
    name = Varchar()
    user = ForeignKey(BaseUser)
```

This was also possible using tags, but was less convenient. Thanks to @sinisaos for reporting this issue.

17.4 0.62.3

Fixed the error message in `LazyTableReference`.

Fixed a bug with `create_pydantic_model` with nested models. For example:

```
create_pydantic_model(Band, nested=(Band.manager,))
```

Sometimes Pydantic couldn't uniquely identify the nested models. Thanks to @wmshort and @sinisaos for their help with this.

17.5 0.62.2

Added a max password length to the `BaseUser` table. By default it's set to 128 characters.

17.6 0.62.1

Fixed a bug with `Readable` when it contains lots of joins.

`Readable` is used to create a user friendly representation of a row in Piccolo Admin.

17.7 0.62.0

Added Many-To-Many support.

```
from piccolo.columns.column_types import (
    ForeignKey,
    LazyTableReference,
    Varchar
)
from piccolo.columns.m2m import M2M
```

(continues on next page)

(continued from previous page)

```
class Band(Table):
    name = Varchar()
    genres = M2M(LazyTableReference("GenreToBand", module_path=__name__))


class Genre(Table):
    name = Varchar()
    bands = M2M(LazyTableReference("GenreToBand", module_path=__name__))


# This is our joining table:
class GenreToBand(Table):
    band = ForeignKey(Band)
    genre = ForeignKey(Genre)


>>> await Band.select(Band.name, Band.genres(Genre.name, as_list=True))
[
  {
    "name": "Pythonistas",
    "genres": ["Rock", "Folk"]
  },
  ...
]
```

See the docs for more details.

Many thanks to @sinisaos and @yezz123 for all the input.

17.8 0.61.2

Fixed some edge cases where migrations would fail if a column name clashed with a reserved Postgres keyword (for example `order` or `select`).

We now have more robust tests for `piccolo asgi new` - as part of our CI we actually run the generated ASGI app to make sure it works (thanks to @AliSayyah and @yezz123 for their help with this).

We also improved docstrings across the project.

17.9 0.61.1

17.9.1 Nicer ASGI template

When using `piccolo asgi new` to generate a web app, it now has a nicer home page template, with improved styles.

17.9.2 Improved schema generation

Fixed a bug with `piccolo schema generate` where it would crash if the column type was unrecognised, due to failing to parse the column's default value. Thanks to @gmos for reporting this issue, and figuring out the fix.

17.9.3 Fix Pylance error

Added `start_connection_pool` and `close_connection_pool` methods to the base `Engine` class (courtesy @gmos).

17.10 0.61.0

The `save` method now supports a `columns` argument, so when updating a row you can specify which values to sync back. For example:

```
band = await Band.objects().get(Band.name == "Pythonistas")
band.name = "Super Pythonistas"
await band.save([Band.name])

# Alternatively, strings are also supported:
await band.save(['name'])
```

Thanks to @trondhindenes for suggesting this feature.

17.11 0.60.2

Fixed a bug with `asyncio.gather` not working with some query types. It was due to them being dataclasses, and they couldn't be hashed properly. Thanks to @brnosouza for reporting this issue.

17.12 0.60.1

Modified the import path for `MigrationManager` in migration files. It was confusing Pylance (VSCode's type checker). Thanks to @gmos for reporting and investigating this issue.

17.13 0.60.0

17.13.1 Secret columns

All column types can now be secret, rather than being limited to the `Secret` column type which is a `Varchar` under the hood (courtesy @sinisaos).

```
class Manager(Table):
    name = Varchar()
    net_worth = Integer(secret=True)
```

The reason this is useful is you can do queries such as:

```
>>> Manager.select(exclude_secrets=True).run_sync()
[{'id': 1, 'name': 'Guido'}]
```

In the Piccolo API project we have `PiccoloCRUD` which is an incredibly powerful way of building an API with very little code. `PiccoloCRUD` has an `exclude_secrets` option which lets you safely expose your data without leaking sensitive information.

17.13.2 Pydantic improvements

max_recursion_depth

`create_pydantic_model` now has a `max_recursion_depth` argument, which is useful when using `nested=True` on large database schemas.

```
>>> create_pydantic_model(MyTable, nested=True, max_recursion_depth=3)
```

Nested tuple

You can now pass a tuple of columns as the argument to `nested`:

```
>>> create_pydantic_model(Band, nested=(Band.manager,))
```

This gives you more control than just using `nested=True`.

include_columns / exclude_columns

You can now include / exclude columns from related tables. For example:

```
>>> create_pydantic_model(Band, nested=(Band.manager,), exclude_columns=(Band.manager.
↪country))
```

Similarly:

```
>>> create_pydantic_model(Band, nested=(Band.manager,), include_columns=(Band.name, Band.
↪manager.name))
```

17.14 0.59.0

- When using `piccolo asgi new` to generate a FastAPI app, the generated code is now cleaner. It also contains a `conftest.py` file, which encourages people to use `piccolo tester run` rather than using `pytest` directly.
- Tidied up docs, and added logo.
- Clarified the use of the `PICCOLO_CONF` environment variable in the docs (courtesy @theelderbeever).
- `create_pydantic_model` now accepts an `include_columns` argument, in case you only want a few columns in your model, it's faster than using `exclude_columns` (courtesy @sinisaos).
- Updated linters, and fixed new errors.

17.15 0.58.0

17.15.1 Improved Pydantic docs

The Pydantic docs used to be in the Piccolo API repo, but have been moved over to this repo. We took this opportunity to improve them significantly with additional examples. Courtesy @sinisaos.

17.15.2 Internal code refactoring

Some of the code has been optimised and cleaned up. Courtesy @yezz123.

17.15.3 Schema generation for recursive foreign keys

When using `piccolo schema generate`, it would get stuck in a loop if a table had a foreign key column which referenced itself. Thanks to @knguyen5 for reporting this issue, and @wmshort for implementing the fix. The output will now look like:

```
class Employee(Table):
    name = Varchar()
    manager = ForeignKey("self")
```

17.15.4 Fixing a bug with Alter.add_column

When using the `Alter.add_column` API directly (not via migrations), it would fail with foreign key columns. For example:

```
SomeTable.alter().add_column(
    name="my_fk_column",
    column=ForeignKey(SomeOtherTable)
  ).run_sync()
```

This has now been fixed. Thanks to @wmshort for discovering this issue.

17.15.5 create_pydantic_model improvements

Additional fields can now be added to the Pydantic schema. This is useful when using Pydantic's JSON schema functionality:

```
my_model = create_pydantic_model(Band, my_extra_field="Hello")
>>> my_model.schema()
{..., "my_extra_field": "Hello"}
```

This feature was added to support new features in Piccolo Admin.

17.15.6 Fixing a bug with import clashes in migrations

In certain situations it was possible to create a migration file with clashing imports. For example:

```
from uuid import UUID
from piccolo.columns.column_types import UUID
```

Piccolo now tries to detect these clashes, and prevent them. If they can't be prevented automatically, a warning is shown to the user. Courtesy @0scarB.

17.16 0.57.0

Added Python 3.10 support (courtesy @kennethcheo).

17.17 0.56.0

17.17.1 Fixed schema generation bug

When using `piccolo schema generate` to auto generate Piccolo `Table` classes from an existing database, it would fail in this situation:

- A table has a column with an index.
- The column name clashed with a Postgres type.

For example, we couldn't auto generate this `Table` class:

```
class MyTable(Table):
    time = Timestamp(index=True)
```

This is because `time` is a builtin Postgres type, and the `CREATE INDEX` statement being inspected in the database wrapped the column name in quotes, which broke our regex.

Thanks to @knguyen5 for fixing this.

17.17.2 Improved testing docs

A convenience method called `get_table_classes` was added to `Finder`.

`Finder` is the main class in Piccolo for dynamically importing projects / apps / tables / migrations etc.

`get_table_classes` lets us easily get the `Table` classes for a project. This makes writing unit tests easier, when we need to setup a schema.

```
from unittest import TestCase

from piccolo.table import create_tables, drop_tables
from piccolo.conf.apps import Finder

TABLES = Finder().get_table_classes()

class TestApp(TestCase):
    def setUp(self):
        create_tables(*TABLES)

    def tearDown(self):
        drop_tables(*TABLES)

    def test_app(self):
        # Do some testing ...
        pass
```

The docs were updated to reflect this.

When dropping tables in a unit test, remember to use `piccolo tester run`, to make sure the test database is used.

17.17.3 get_output_schema

`get_output_schema` is the main entrypoint for database reflection in Piccolo. It has been modified to accept an optional `Engine` argument, which makes it more flexible.

17.18 0.55.0

17.18.1 Table._meta.refresh_db

Added the ability to refresh the database engine.

```
MyTable._meta.refresh_db()
```

This causes the `Table` to fetch the `Engine` again from your `piccolo_conf.py` file. The reason this is useful, is you might change the `PICCOLO_CONF` environment variable, and some `Table` classes have already imported an engine. This is now used by the `piccolo tester run` command to ensure all `Table` classes have the correct engine.

17.18.2 ColumnMeta edge cases

Fixed an edge case where `ColumnMeta` couldn't be copied if it had extra attributes added to it.

17.18.3 Improved column type conversion

When running migrations which change column types, Piccolo now provides the `USING` clause to the `ALTER COLUMN` DDL statement, which makes it more likely that type conversion will be successful.

For example, if there is an `Integer` column, and it's converted to a `Varchar` column, the migration will run fine. In the past, running this in reverse would fail. Now Postgres will try and cast the values back to integers, which makes reversing migrations more likely to succeed.

17.18.4 Added drop_tables

There is now a convenience function for dropping several tables in one go. If the database doesn't support `CASCADE`, then the tables are sorted based on their `ForeignKey` columns, so they're dropped in the correct order. It all runs inside a transaction.

```
from piccolo.table import drop_tables

drop_tables(Band, Manager)
```

This is a useful tool in unit tests.

17.18.5 Index support in schema generation

When using `piccolo schema generate`, Piccolo will now reflect the indexes from the database into the generated `Table` classes. Thanks to @wmshort for this.

17.19 0.54.0

Added the `db_column_name` option to columns. This is for edge cases where a legacy database is being used, with problematic column names. For example, if a column is called `class`, this clashes with a Python builtin, so the following isn't possible:

```
class MyTable(Table):
    class = Varchar()  # Syntax error!
```

You can now do the following:

```
class MyTable(Table):
    class_ = Varchar(db_column_name='class')
```

Here are some example queries using it:

```
# Create - both work as expected
MyTable(class_='Test').save().run_sync()
MyTable.objects().create(class_='Test').run_sync()

# Objects
row = MyTable.objects().first().where(MyTable.class_ == 'Test').run_sync()
>>> row.class_
'Test'

# Select
>>> MyTable.select().first().where(MyTable.class_ == 'Test').run_sync()
{'id': 1, 'class': 'Test'}
```

17.20 0.53.0

An internal code clean up (courtesy @yezz123).

Dramatically improved CLI appearance when running migrations (courtesy @wmshort).

Added a runtime reflection feature, where `Table` classes can be generated on the fly from existing database tables (courtesy @AliSayyah). This is useful when dealing with very dynamic databases, where tables are frequently being added / modified, so hard coding them in a `tables.py` is impractical. Also, for exploring databases on the command line. It currently just supports Postgres.

Here's an example:

```
from piccolo.table_reflection import TableStorage

storage = TableStorage()
Band = await storage.get_table('band')
>>> await Band.select().run()
[{'id': 1, 'name': 'Pythonistas', 'manager': 1}, ...]
```

17.21 0.52.0

Lots of improvements to `piccolo schema generate`:

- Dramatically improved performance, by executing more queries in parallel (courtesy @AliSayyah).
- If a table in the database has a foreign key to a table in another schema, this will now work (courtesy @AliSayyah).
- The column defaults are now extracted from the database (courtesy @wmshort).
- The `scale` and `precision` values for `Numeric` / `Decimal` column types are extracted from the database (courtesy @wmshort).
- The `ON DELETE` and `ON UPDATE` values for `ForeignKey` columns are now extracted from the database (courtesy @wmshort).

Added `BigSerial` column type (courtesy @aliereno).

Added GitHub issue templates (courtesy @AbhijithGanesh).

17.22 0.51.1

Fixing a bug with `on_delete` and `on_update` not being set correctly. Thanks to @wmshort for discovering this.

17.23 0.51.0

Modified `create_pydantic_model`, so `JSON` and `JSONB` columns have a `format` attribute of `'json'`. This will be used by Piccolo Admin for improved JSON support. Courtesy @sinisaos.

Fixing a bug where the `piccolo fixtures load` command wasn't registered with the Piccolo CLI.

17.24 0.50.0

The `where` clause can now accept multiple arguments (courtesy @AliSayyah):

```
Concert.select().where(
    Concert.venue.name == 'Royal Albert Hall',
    Concert.band_1.name == 'Pythonistas'
).run_sync()
```

It's another way of expressing *AND*. It's equivalent to both of these:

```
Concert.select().where(
    Concert.venue.name == 'Royal Albert Hall'
).where(
    Concert.band_1.name == 'Pythonistas'
).run_sync()

Concert.select().where(
    (Concert.venue.name == 'Royal Albert Hall') & (Concert.band_1.name == 'Pythonistas')
).run_sync()
```

Added a `create` method, which is an easier way of creating objects (courtesy @AliSayyah).

```
# This still works:
band = Band(name="C-Sharps", popularity=100)
band.save().run_sync()

# But now we can do it in a single line using `create`:
band = Band.objects().create(name="C-Sharps", popularity=100).run_sync()
```

Fixed a bug with `piccolo schema generate` where columns with unrecognised column types were omitted from the output (courtesy @AliSayyah).

Added docs for the `--trace` argument, which can be used with Piccolo commands to get a traceback if the command fails (courtesy @hipertracker).

Added `DoublePrecision` column type, which is similar to `Real` in that it stores `float` values. However, those values are stored at greater precision (courtesy @AliSayyah).

Improved `AppRegistry`, so if a user only adds the app name (e.g. `blog`), instead of `blog.piccolo_app`, it will now emit a warning, and will try to import `blog.piccolo_app` (courtesy @aliereno).

17.25 0.49.0

Fixed a bug with `create_pydantic_model` when used with a `Decimal` / `Numeric` column when no `digits` arguments was set (courtesy @AliSayyah).

Added the `create_tables` function, which accepts a sequence of `Table` subclasses, then sorts them based on their `ForeignKey` columns, and creates them. This is really useful for people who aren't using migrations (for example, when using Piccolo in a simple data science script). Courtesy @AliSayyah.

```
from piccolo.tables import create_tables

create_tables(Band, Manager, if_not_exists=True)

# Equivalent to:
Manager.create_table(if_not_exists=True).run_sync()
Band.create_table(if_not_exists=True).run_sync()
```

Fixed typos with the new fixtures app - sometimes it was referred to as `fixture` and other times `fixtures`. It's now standardised as `fixtures` (courtesy @hipertracker).

17.26 0.48.0

The `piccolo user create` command can now be used by passing in command line arguments, instead of using the interactive prompt (courtesy @AliSayyah).

For example `piccolo user create --username=bob ...`.

This is useful when you want to create users in a script.

17.27 0.47.0

You can now use `pip install piccolo[all]`, which will install all optional requirements.

17.28 0.46.0

Added the fixtures app. This is used to dump data from a database to a JSON file, and then reload it again. It's useful for seeding a database with essential data, whether that's a colleague setting up their local environment, or deploying to production.

To create a fixture:

```
piccolo fixtures dump --apps=blog > fixture.json
```

To load a fixture:

```
piccolo fixtures load fixture.json
```

As part of this change, Piccolo's Pydantic support was brought into this library (prior to this it only existed within the `piccolo_api` library). At a later date, the `piccolo_api` library will be updated, so it's Pydantic code just proxies to what's within the main `piccolo` library.

17.29 0.45.1

Improvements to `piccolo schema generate`. It's now smarter about which imports to include. Also, the `Table` classes output will now be sorted based on their `ForeignKey` columns. Internally the sorting algorithm has been changed to use the `graphlib` module, which was added in Python 3.9.

17.30 0.45.0

Added the `piccolo schema graph` command for visualising your database structure, which outputs a Graphviz file. It can then be turned into an image, for example:

```
piccolo schema map | dot -Tpdf -o graph.pdf
```

Also made some minor changes to the ASGI templates, to reduce MyPy errors.

17.31 0.44.1

Updated `to_dict` so it works with nested objects, as introduced by the `prefetch` functionality.

For example:

```
band = Band.objects(Band.manager).first().run_sync()

>>> band.to_dict()
{'id': 1, 'name': 'Pythonistas', 'manager': {'id': 1, 'name': 'Guido'}}
```

It also works with filtering:

```
>>> band.to_dict(Band.name, Band.manager.name)
{'name': 'Pythonistas', 'manager': {'name': 'Guido'}}
```

17.32 0.44.0

Added the ability to prefetch related objects. Here's an example:

```
band = await Band.objects(Band.manager).run()
>>> band.manager
<Manager: 1>
```

If a table has a lot of `ForeignKey` columns, there's a useful shortcut, which will return all of the related rows as objects.

```
concert = await Concert.objects(Concert.all_related()).run()
>>> concert.band_1
<Band: 1>
>>> concert.band_2
<Band: 2>
>>> concert.venue
<Venue: 1>
```

Thanks to @wmshort for all the input.

17.33 0.43.0

Migrations containing `Array`, `JSON` and `JSONB` columns should be more reliable now. More unit tests were added to cover edge cases.

17.34 0.42.0

You can now use `all_columns` at the root. For example:

```
await Band.select(
    Band.all_columns(),
    Band.manager.all_columns()
).run()
```

You can also exclude certain columns if you like:

```
await Band.select(
    Band.all_columns(exclude=[Band.id]),
    Band.manager.all_columns(exclude=[Band.manager.id])
).run()
```

17.35 0.41.1

Fix a regression where if multiple tables are created in a single migration file, it could potentially fail by applying them in the wrong order.

17.36 0.41.0

Fixed a bug where if `all_columns` was used two or more levels deep, it would fail. Thanks to @wmshort for reporting this issue.

Here's an example:

```
Concert.select(
    Concert.venue.name,
    *Concert.band_1.manager.all_columns()
).run_sync()
```

Also, the `ColumnsDelegate` has now been tweaked, so unpacking of `all_columns` is optional.

```
# This now works the same as the code above (we have omitted the *)
Concert.select(
    Concert.venue.name,
    Concert.band_1.manager.all_columns()
).run_sync()
```

17.37 0.40.1

Loosen the `typing-extensions` requirement, as it was causing issues when installing `asyncpg`.

17.38 0.40.0

Added `nested` output option, which makes the response from a `select` query use nested dictionaries:

```
>>> await Band.select(Band.name, *Band.manager.all_columns()).output(nested=True).run()
[{'name': 'Pythonistas', 'manager': {'id': 1, 'name': 'Guido'}}]
```

Thanks to @wmshort for the idea.

17.39 0.39.0

Added `to_dict` method to `Table`.

If you just use `__dict__` on a `Table` instance, you get some non-column values. By using `to_dict` it's just the column values. Here's an example:

```
class MyTable(Table):
    name = Varchar()

instance = MyTable.objects().first().run_sync()

>>> instance.__dict__
{'_exists_in_db': True, 'id': 1, 'name': 'foo'}

>>> instance.to_dict()
{'id': 1, 'name': 'foo'}
```

Thanks to @wmshort for the idea, and @aminalaee and @sinisaos for investigating edge cases.

17.40 0.38.2

Removed problematic type hint which assumed pytest was installed.

17.41 0.38.1

Minor changes to `get_or_create` to make sure it handles joins correctly.

```
instance = (
    Band.objects()
    .get_or_create(
        (Band.name == "My new band")
        & (Band.manager.name == "Excellent manager")
    )
    .run_sync()
)
```

In this situation, there are two columns called `name` - we need to make sure the correct value is applied if the row doesn't exist.

17.42 0.38.0

`get_or_create` now supports more complex where clauses. For example:

```
row = await Band.objects().get_or_create(
    (Band.name == 'Pythonistas') & (Band.popularity == 1000)
).run()
```

And you can find out whether the row was created or not using `row._was_created`.

Thanks to @wmshort for reporting this issue.

17.43 0.37.0

Added `ModelBuilder`, which can be used to generate data for tests (courtesy @aminalaee).

17.44 0.36.0

Fixed an issue where `like` and `ilike` clauses required a wildcard. For example:

```
await Manager.select().where(Manager.name.ilike('Guido%')).run()
```

You can now omit wildcards if you like:

```
await Manager.select().where(Manager.name.ilike('Guido')).run()
```

Which would match on `'guido'` and `'Guido'`, but not `'Guidoxyz'`.

Thanks to @wmshort for reporting this issue.

17.45 0.35.0

- Improved `PrimaryKey` deprecation warning (courtesy @tonybaloney).
- Added `piccolo schema generate` which creates a Piccolo schema from an existing database.
- Added `piccolo tester run` which is a wrapper around pytest, and temporarily sets `PICCOLO_CONF`, so a test database is used.
- Added the `get` convenience method (courtesy @aminalaee). It returns the first matching record, or `None` if there's no match. For example:

```
manager = await Manager.objects().get(Manager.name == 'Guido').run()

# This is equivalent to:
manager = await Manager.objects().where(Manager.name == 'Guido').
↪first().run()
```

17.46 0.34.0

Added the `get_or_create` convenience method (courtesy @aminalaee). Example usage:

```
manager = await Manager.objects().get_or_create(
    Manager.name == 'Guido'
).run()
```

17.47 0.33.1

- Bug fix, where `compare_dicts` was failing in migrations if any `Column` had an unhashable type as an argument. For example: `Array(default=[])`. Thanks to @hipertracker for reporting this problem.
- Increased the minimum version of orjson, so binaries are available for Macs running on Apple silicon (courtesy @hipertracker).

17.48 0.33.0

Fix for auto migrations when using custom primary keys (thanks to @adriangb and @aminalaee for investigating this issue).

17.49 0.32.0

Migrations can now have a description, which is shown when using `piccolo migrations check`. This makes migrations easier to identify (thanks to @davidolrik for the idea).

17.50 0.31.0

Added an `all_columns` method, to make it easier to retrieve all related columns when doing a join. For example:

```
await Band.select(Band.name, *Band.manager.all_columns()).first().run()
```

Changed the instructions for installing additional dependencies, so they're wrapped in quotes, to make sure it works on ZSH (i.e. `pip install 'piccolo[postgres]'` instead of `pip install piccolo[postgres]`).

17.51 0.30.0

The database drivers are now installed separately. For example: `pip install piccolo[postgres]` (courtesy @aminalaee).

For some users this might be a **breaking change** - please make sure that for existing Piccolo projects, you have either `asyncpg`, or `piccolo[postgres]` in your `requirements.txt` file.

17.52 0.29.0

The user can now specify the primary key column (courtesy @aminalaee). For example:

```
class RecordingStudio(Table):
    pk = UUID(primary_key=True)
```

The BlackSheep template generated by *piccolo asgi new* now supports mounting of the Piccolo Admin (courtesy @sinisaos).

17.53 0.28.0

Added aggregations functions, such as `Sum`, `Min`, `Max` and `Avg`, for use in select queries (courtesy @sinisaos).

17.54 0.27.0

Added uvloop as an optional dependency, installed via *pip install piccolo[uvloop]* (courtesy @aminalaee). uvloop is a faster implementation of the asyncio event loop found in Python's standard library. When uvloop is installed, Piccolo will use it to increase the performance of the Piccolo CLI, and web servers such as Uvicorn will use it to increase the performance of your ASGI app.

17.55 0.26.0

Added `eq` and `ne` methods to the `Boolean` column, which can be used if linters complain about using `SomeTable.some_column == True`.

17.56 0.25.0

- Changed the migration IDs, so the timestamp now includes microseconds. This is to make clashing migration IDs much less likely.
- Added a lot of end-to-end tests for migrations, which revealed some bugs in `Column` defaults.

17.57 0.24.1

A bug fix for migrations. See issue 123 for more information.

17.58 0.24.0

Lots of improvements to `JSON` and `JSONB` columns. Piccolo will now automatically convert between Python types and JSON strings. For example, with this schema:

```
class RecordingStudio(Table):
    name = Varchar()
    facilities = JSON()
```

We can now do the following:

```
RecordingStudio(
    name="Abbey Road",
    facilities={'mixing_desk': True}  # Will automatically be converted to a JSON string
).save().run_sync()
```

Similarly, when fetching data from a JSON column, Piccolo can now automatically deserialise it.

```
>>> RecordingStudio.select().output(load_json=True).run_sync()
[{'id': 1, 'name': 'Abbey Road', 'facilities': {'mixing_desk': True}]

>>> studio = RecordingStudio.objects().first().output(load_json=True).run_sync()
>>> studio.facilities
{'mixing_desk': True}
```

17.59 0.23.0

Added the `create_table_class` function, which can be used to create `Table` subclasses at runtime. This was required to fix an existing bug, which was effecting migrations (see issue 111 for more details).

17.60 0.22.0

- An error is now raised if a user tries to create a Piccolo app using `piccolo app new` with the same name as a builtin Python module, as it will cause strange bugs.
- Fixing a strange bug where using an expression such as `Concert.band_1.manager.id` in a query would cause an error. It only happened if multiple joins were involved, and the last column in the chain was `id`.
- `where` clauses can now accept `Table` instances. For example: `await Band.select().where(Band.manager == some_manager).run()`, instead of having to explicity reference the `id`.

17.61 0.21.2

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

17.62 0.21.1

Fix missing imports in FastAPI and Starlette app templates.

17.63 0.21.0

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

17.64 0.20.0

Added `choices` option to `Column`.

17.65 0.19.1

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

17.66 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
```

17.67 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).

17.68 0.18.3

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

17.69 0.18.2

Added support for filtering `Array` columns.

17.70 0.18.1

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

17.71 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.

17.72 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.

17.73 0.17.4

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

17.74 0.17.3

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

17.75 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.

17.76 0.17.1

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

17.77 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.

17.78 0.16.5

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

17.79 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.

17.80 0.16.3

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

17.81 0.16.2

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

17.82 0.16.1

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

17.83 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.

17.84 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.

17.85 0.15.0

Added `Bytea` / `Blob` column type.

17.86 0.14.13

Fixing a bug with migrations which drop column defaults.

17.87 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.

17.88 0.14.11

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

17.89 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.

17.90 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.

17.91 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.

17.92 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.

17.93 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.

17.94 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

17.95 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.

17.96 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.

17.97 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.

17.98 0.14.1

Fixing a bug with dropping tables in auto migrations.

17.99 0.14.0

Added `Interval` column type.

17.100 0.13.5

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

17.101 0.13.4

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

17.102 0.13.3

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

17.103 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.

17.104 0.13.1

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

17.105 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.

17.106 0.12.6

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

17.107 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.

17.108 0.12.4

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

17.109 0.12.3

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

17.110 0.12.2

Fixing bug when sorting `extra_definitions` in auto migrations.

17.111 0.12.1

- Fixed typos.
- Bumped requirements.

17.112 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.

17.113 0.11.8

ASGI template fix.

17.114 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.

17.115 0.11.6

Added support for FastAPI to `piccolo asgi new`.

17.116 0.11.5

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

17.117 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.

17.118 0.11.3

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

17.119 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.

17.120 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.

17.121 0.11.0

Added `Numeric` and `Real` column types.

17.122 0.10.8

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

17.123 0.10.7

Improving release script.

17.124 0.10.6

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

17.125 0.10.5

Auto migrations can now run backwards.

17.126 0.10.4

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

17.127 0.10.3

Adding missing jinja templates to setup.py.

17.128 0.10.2

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

17.129 0.10.1

Fixing bug with enum default values.

17.130 0.10.0

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

17.131 0.9.3

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

17.132 0.9.2

`.first()` bug fix.

17.133 0.9.1

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

17.134 0.9.0

Added support for auto migrations.

17.135 0.8.3

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

17.136 0.8.2

Fixing release issue.

17.137 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.

17.138 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.

17.139 0.7.7

Column sequencing matches the definition order.

17.140 0.7.6

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

17.141 0.7.5

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

17.142 0.7.4

Bug fixes and dependency updates.

17.143 0.7.3

Adding missing `__int__.py` file.

17.144 0.7.2

Changed migration import paths.

17.145 0.7.1

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

17.146 0.7.0

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

17.147 0.6.1

Adding missing `__init__.py` files.

17.148 0.6.0

Moved `BaseUser`. Migration refactor.

17.149 0.5.2

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

17.150 0.5.1

Added `batch` support.

17.151 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.

17.152 0.4.1

SQLite is now returning datetime objects for timestamp fields.

17.153 0.4.0

Refactored to improve code completion, along with bug fixes.

17.154 0.3.7

Allowing `Update` queries in SQLite.

17.155 0.3.6

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

17.156 0.3.5

Renamed `User` to `BaseUser`.

17.157 0.3.4

Added `ilike`.

17.158 0.3.3

Added value types to columns.

17.159 0.3.2

Default values infer the engine type.

17.160 0.3.1

Update click version.

17.161 0.3

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

17.162 0.2

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

17.163 0.1.2

Allowing joins across multiple tables.

17.164 0.1.1

Added playground.

CHAPTER

EIGHTEEN

HELP

If you have any questions then the best place to ask them is the discussions section on our GitHub page.

CHAPTER

NINETEEN

TLDR

Install Piccolo:

```
pip install piccolo
```

Experiment with queries:

```
piccolo playground run
```

Give me an ASGI web app!

```
piccolo asgi new
```

FastAPI, Starlette, and BlackSheep are currently supported, with more coming soon.

CHAPTER

TWENTY

VIDEOS

Piccolo has some tutorial videos on YouTube, which are a great companion to the docs.

INDEX

Symbols

__getitem__() (*piccolo.columns.column_types.Array method*), 55

A

add_m2m() (*piccolo.table.Table method*), 58
all() (*piccolo.columns.column_types.Array method*), 56
any() (*piccolo.columns.column_types.Array method*), 56
AppRegistry (*class in piccolo.conf.apps*), 66
Array (*class in piccolo.columns.column_types*), 55

B

BigInt (*class in piccolo.columns.column_types*), 47
BigSerial (*class in piccolo.columns.column_types*), 47
Boolean (*class in piccolo.columns.column_types*), 44
Bytea (*class in piccolo.columns.column_types*), 43

C

Column (*class in piccolo.columns.column_types*), 42
Count (*class in piccolo.query.methods.select*), 32
create_pydantic_model() (*in module piccolo.utils.pydantic*), 93

D

Date (*class in piccolo.columns.column_types*), 51
DoublePrecision (*class in piccolo.columns.column_types*), 48

F

ForeignKey (*class in piccolo.columns.column_types*), 44
freeze() (*piccolo.query.base.Query method*), 40

G

get_m2m() (*piccolo.table.Table method*), 59

I

Integer (*class in piccolo.columns.column_types*), 48
Interval (*class in piccolo.columns.column_types*), 52

J

JSON (*class in piccolo.columns.column_types*), 54
JSONB (*class in piccolo.columns.column_types*), 54

N

Numeric (*class in piccolo.columns.column_types*), 48

P

PostgresEngine (*class in piccolo.engine.postgres*), 79

R

Real (*class in piccolo.columns.column_types*), 49
remove_m2m() (*piccolo.table.Table method*), 59

S

Secret (*class in piccolo.columns.column_types*), 50
Serial (*class in piccolo.columns.column_types*), 49
SmallInt (*class in piccolo.columns.column_types*), 49
SQLiteEngine (*class in piccolo.engine.sqlite*), 77

T

table_finder() (*in module piccolo.conf.apps*), 69
Text (*class in piccolo.columns.column_types*), 50
Time (*class in piccolo.columns.column_types*), 52
Timestamp (*class in piccolo.columns.column_types*), 53
Timestamptz (*class in piccolo.columns.column_types*), 53

U

UUID (*class in piccolo.columns.column_types*), 50

V

Varchar (*class in piccolo.columns.column_types*), 51