
Piccolo

Release 1.34.0

Daniel Townsend

May 26, 2026

CONTENTS:

1	Getting Started	3
2	Query Types	11
3	Query Clauses	41
4	Functions	63
5	Schema	75
6	Projects and Apps	107
7	Engines	117
8	Migrations	125
9	Authentication	133
10	ASGI	137
11	Serialization	139
12	Testing	145
13	Features	151
14	Playground	153
15	Ecosystem	155
16	Tutorials	157
17	Contributing	167
18	Changes	169
19	Help	243
20	API reference	245
21	About Piccolo v1	263
22	TLDR	265

23 Videos

267

Index

269

i Note

These are the docs for **Piccolo v1**. *Read more here.*

For v0.x docs [go here](#).

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

GETTING STARTED

1.1 What is Piccolo?

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

Some of its stand out features are:

- Support for *sync and async*.
- A builtin *playground*, which makes learning a breeze.
- Fully type annotated, with great *tab completion support* - it works great with *iPython* and *VSCode*.
- Batteries included - a *User model and authentication, migrations, an admin*, and more.
- Templates for creating your own *ASGI web app*.

1.1.1 History

Piccolo was created while working at a design agency, where almost all projects being undertaken were API driven (often with high traffic), and required web sockets. The author was naturally interested in the possibilities of *asyncio*. Piccolo is built from the ground up with *asyncio* in mind. Likewise, Piccolo makes extensive use of *type annotations*, another innovation in Python around the time Piccolo was started.

A really important thing when working at a design agency is having a **great admin interface**. A huge amount of effort has gone into *Piccolo Admin* to make something you'd be proud to give to a client.

A lot of batteries are included because Piccolo is a pragmatic framework focused on delivering quality, functional apps to customers. This is why we have templating tools like *piccolo asgi new* for getting a web app started quickly, automatic database migrations for making iteration fast, and lots of authentication middleware and endpoints for rapidly *building APIs* out of the box.

Piccolo has been used extensively by the author on professional projects, for a range of corporate and startup clients.

1.2 Database Support

Postgres is the primary database which Piccolo was designed for. It's robust, feature rich, and a great choice for most projects.

CockroachDB is also supported. It's designed to be scalable and fault tolerant, and is mostly compatible with *Postgres*. There may be some minor features not supported, but it's OK to use.

SQLite support was originally added to enable tooling like the *playground*, but over time we've added more and more support. Many people successfully use *SQLite* and Piccolo together in production. The main missing feature is support for *automatic database migrations* due to *SQLite*'s limited support for `ALTER TABLE` DDL statements.

1.2.1 What about other databases?

Our focus is on providing great support for a limited number of databases (especially Postgres), however it's likely that we'll support more databases in the future.

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 or CockroachDB 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]'
```

Hint

On Windows, you may need to use double quotes instead. For example `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, 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 or Cockroach, 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:

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

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](#)).

1.6 Setup Cockroach

1.6.1 Installation

Follow the [instructions](#) for your OS.

Versions

We support the latest stable version.

Note

Features using `format()` will be available in v22.2 or higher, but we recommend using the stable version so you can upgrade automatically when it becomes generally available.

Cockroach is designed to be a “rolling database”: Upgrades are as simple as switching out to the next version of a binary (or changing a number in a `docker-compose.yml`). This has one caveat: You cannot upgrade an “alpha” release. It is best to stay on the latest stable.

1.6.2 Creating a database

cockroach sql

CockroachDB comes with its own management tooling.

```
cd ~/wherever/you/installed/cockroachdb
cockroach sql --insecure
```

Enter the following:

```
create database piccolo;
use piccolo;
```

Management GUI

CockroachDB comes with its own web-based management GUI available on localhost: <http://127.0.0.1:8080/>

Beekeeper Studio

If you prefer a GUI, Beekeeper Studio is recommended and has an [installer available](#).

1.6.3 Column Types

As of this writing, CockroachDB will always convert JSON to JSONB and will always report INTEGER as BIGINT.

Piccolo will automatically handle these special cases for you, but we recommend being explicit about this to prevent complications in future versions of Piccolo.

- Use `JSONB()` instead of `JSON()`
- Use `BigInt()` instead of `Integer()`

1.7 Setup SQLite

1.7.1 Installation

The good news is SQLite is good to go out of the box with Python.

Some Piccolo features are only available with newer SQLite versions.

1.7.2 Check version

To check which SQLite version you're using, simply open a Python terminal, and do the following:

```
>>> import sqlite3
>>> sqlite3.sqlite_version
'3.39.0'
```

The easiest way to upgrade your SQLite version is to install the latest version of Python.

1.8 Example Schema

This is the schema used by the example queries throughout the docs, and also in the *playground*.

Manager and Band are most commonly used:

```
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()
```

We sometimes use these other tables in the examples too:

```
class Venue(Table):
    name = Varchar()
    capacity = Integer()

class Concert(Table):
    band_1 = ForeignKey(references=Band)
    band_2 = ForeignKey(references=Band)
    venue = ForeignKey(references=Venue)
    starts = Timestamp()
    duration = Interval()
```

(continues on next page)

(continued from previous page)

```
class Ticket(Table):
    concert = ForeignKey(references=Concert)
    price = Numeric()

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

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

1.9 Sync and Async

One of the motivations for making Piccolo was the lack of ORMs and query builders which support asyncio.

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

1.9.1 Async example

You can await a query to run it:

```
>>> await Band.select(Band.name)
[{'name': 'Pythonistas'}]
```

Alternatively, you can await a query's run method:

```
# This makes it extra explicit that a database query is being made:
>>> await Band.select(Band.name).run()

# It also gives you more control over how the query is run.
# For example, if we wanted to bypass the connection pool for some reason:
>>> await Band.select(Band.name).run(in_pool=False)
```

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

1.9.2 Sync example

This lets you execute a query in an application which isn't using asyncio:

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

1.9.3 Explicit

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

Until you execute `await` or `run_sync`, you can chain as many methods onto your query as you like, safe in the knowledge that no database queries are being made.

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*.

2.1.1 Columns

To get all columns:

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

To get certain columns:

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

Or use an alias to make it shorter:

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

Hint

All of these examples also work synchronously using `run_sync` - see *Sync and Async*.

2.1.2 as_alias

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

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

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

2.1.3 Joins

One of the most powerful things about `select` is its support for joins.

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

The joins can go several layers deep.

```
>>> await Concert.select(Concert.id, Concert.band_1.manager.name)
[{'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:

```
>>> await Band.select(Band.name, Band.manager.all_columns())
[
  {'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:

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

You can exclude some columns if you like:

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

Strings are supported too if you prefer:

```
>>> await Band.select(
...     Band.name,
...     Band.manager.all_columns(exclude=['id'])
... )
[
  {'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:

```
>>> await Band.select(
...     Band.all_columns(exclude=[Band.id]),
...     Band.manager.all_columns(exclude=[Band.manager.id])
... )
[
  {'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:

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

2.1.4 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.

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

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

2.1.5 String functions

Piccolo has lots of string functions built-in. See `piccolo/query/functions/string.py`. Here's an example using `Upper`, to convert values to uppercase:

```
from piccolo.query.functions.string import Upper

>> await Band.select(Upper(Band.name, alias='name'))
[{'name': 'PYTHONISTAS'}, ...]
```

You can also use these within where clauses:

```
from piccolo.query.functions.string import Upper

>> await Band.select(Band.name).where(Upper(Band.manager.name) == 'GUIDO')
[{'name': 'Pythonistas'}]
```

2.1.6 Aggregate functions

Note

These can all be used in conjunction with the *group_by* clause.

Count

Hint

You can use the *count* query as a quick way of getting the number of rows in a table.

Returns the number of matching rows.

```
from piccolo.query.functions.aggregate import Count

>> await Band.select(Count()).where(Band.popularity > 100)
[{'count': 3}]
```

To find out more about the options available, see *Count*.

Avg

Returns the average for a given column:

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

Sum

Returns the sum for a given column:

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

Max

Returns the maximum for a given column:

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

Min

Returns the minimum for a given column:

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

Additional features

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

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

And can use aliases for aggregate functions like this:

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

2.1.7 SelectRaw

In certain situations you may want to have raw SQL in your select query.

For example, if there's a Postgres function which you want to access, which isn't supported by Piccolo:

```
from piccolo.query import SelectRaw

>>> await Band.select(
...     Band.name,
...     SelectRaw("log(popularity) AS log_popularity")
... )
[{'name': 'Pythonistas', 'log_popularity': 3.0}]
```

 **Warning**

Only use SQL that you trust.

2.1.8 Query clauses

batch

See *batch*.

callback

See *callback*.

columns

By default all columns are returned from the queried table.

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

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

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

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

distinct

See *distinct*.

first

See *first*.

group_by

See *group_by*.

limit

See *limit*.

lock_rows

See *lock_rows*.

offset

See *offset*.

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 rows:

```
>>> await Band.objects()
[<Band: 1>, <Band: 2>, <Band: 3>]
```

To limit the number of rows returned, use the *order_by* and *limit* clauses:

```
>>> await Band.objects().order_by(Band.popularity, ascending=False).limit(2)
[<Band: 1>, <Band: 3>]
```

To filter the rows we use the *where* clause:

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

To get a single row (or `None` if it doesn't exist) use the *first* clause:

```
>>> await Band.objects().where(Band.name == 'Pythonistas').first()
<Band: 1>
```

Alternatively, you can use this abbreviated syntax:

```
>>> await Band.objects().get(Band.name == 'Pythonistas')
<Band: 1>
```

You'll notice that the API is similar to *Select* (expect with `select` you can specify which columns are returned).

2.2.2 Creating objects

You can pass the column values using kwargs:

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

Alternatively, you can pass in a dictionary, which is friendlier to static analysis tools like Mypy (it can easily detect typos in the column names):

```
>>> band = Band({Band.name: "C-Sharps", Band.popularity: 100})
>>> await band.save()
```

We also have this shortcut which combines the above into a single line:

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

2.2.3 Updating objects

save

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

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

band.popularity = 100000

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

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

update_self

The *save* method is fine in the majority of cases, but there are some situations where the *update_self* method is preferable.

For example, if we want to increment the popularity value, we can do this:

```
await band.update_self({
    Band.popularity: Band.popularity + 1
})
```

Which does the following:

- Increments the popularity in the database
- Assigns the new value to the object

This is safer than:

```
band.popularity += 1
await band.save()
```

Because `update_self` increments the current popularity value in the database, not the one on the object, which might be out of date.

2.2.4 Deleting objects

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

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

await band.remove()
```

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 = await Band.objects().where(
    Band.name == 'Pythonistas'
).first()

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

It works multiple levels deep - for example:

```
concert = await Concert.objects().first()
manager = await concert.get_related(Concert.band_1.manager)
```

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 = await Band.objects(Band.manager).where(
    Band.name == 'Pythonistas'
).first()

>>> band.manager
<Manager: 1>
```

(continues on next page)

(continued from previous page)

```
>>> 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 = await Ticket.objects(
    Ticket.concert.all_related()
).first()

# Any intermediate objects will also be loaded:
>>> ticket.concert
<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'
await ticket.concert.band_1.save()
```

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

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

ticket = await Ticket.objects().prefetch(
    Ticket.concert.all_related()
).first()
```

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

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

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

```
band = await Band.objects.get_or_create(
    Band.name == 'Pythonistas'
)
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 = await Band.objects().get_or_create(
    (Band.name == 'Pythonistas') & (Band.popularity == 1000),
)

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

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 = await Band.objects().first()

>>> 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 = await Band.objects().first()

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

2.2.8 refresh

If you have an object which has gotten stale, and want to refresh it, so it has the latest data from the database, you can use the `refresh` method.

```
# If we have an instance:
band = await Band.objects().first()

# And it has gotten stale, we can refresh it:
await band.refresh()

# Or just refresh certain columns:
await band.refresh([Band.name])
```

It works with prefetch too:

```
# If we have an instance with a child object:
band = await Band.objects(Band.manager).first()

# And it has gotten stale, we can refresh it:
await band.refresh()

# The nested object will also be updated if it was stale:
>>> band.manager.name
"New value"
```

refresh is very useful in unit tests:

```
# If we have an instance:
band = await Band.objects().where(Band.name == "Pythonistas").first()

# Call an API endpoint which updates the object (e.g. with httpx):
await client.patch(f"/band/{band.id}/", json={"popularity": 5000})

# Make sure the instance was updated:
await band.refresh()
assert band.popularity == 5000
```

2.2.9 Comparing objects

If you have two objects, and you want to know whether they refer to the same row in the database, you can simply use the equality operator:

```
band_1 = await Band.objects().where(Band.name == "Pythonistas").first()
band_2 = await Band.objects().where(Band.name == "Pythonistas").first()

>>> band_1 == band_2
True
```

It works by comparing the primary key value of each object. It's equivalent to this:

```
>>> band_1.id == band_2.id
True
```

If the object has no primary key value yet (e.g. it uses a `Serial` column, and it hasn't been saved in the database), then the result will always be `False`:

```
band_1 = Band()
band_2 = Band()

>>> band_1 == band_2
False
```

If you want to compare every value on the objects, and not just the primary key, you can use `to_dict`. For example:

```
>>> band_1.to_dict() == band_2.to_dict()
True
```

(continues on next page)

(continued from previous page)

```
>>> band_1.popularity = 10_000
>>> band_1.to_dict() == band_2.to_dict()
False
```

2.2.10 Query clauses

batch

See *batch*.

callback

See *callback*.

first

See *first*.

limit

See *limit*.

lock_rows

See *lock_rows*.

offset

See *offset*.

order_by

See *order_by*.

output

See *output*.

where

See *where*.

2.3 Count

The count query makes it really easy to retrieve the number of rows in a table:

```
>>> await Band.count()
3
```

It's equivalent to this select query:

```

from piccolo.query.functions.aggregate import Count

>>> response = await Band.select(Count())
>>> response[0]['count']
3

```

As you can see, the count query is more convenient.

2.3.1 Non-null columns

If you want to retrieve the number of rows where a given column isn't null, we can do so as follows:

```

await Band.count(column=Band.name)

# Or simply:
await Band.count(Band.name)

```

Note, this is equivalent to:

```

await Band.count().where(Band.name.is_not_null())

```

Example

If we have the following database table:

```

class Band(Table):
    name = Varchar()
    popularity = Integer(null=True)

```

With the following data:

name	popularity
Pythonistas	1000
Rustaceans	800
C-Sharps	null

Then we get the following results:

```

>>> await Band.count()
3

>>> await Band.count(Band.popularity)
2

```

2.3.2 distinct

We can count the number of distinct (i.e. unique) rows.

```

await Band.count(distinct=[Band.name])

# This also works - use whichever you prefer:
await Band.count().distinct([Band.name])

```

With the following data:

name	popularity
Pythonistas	1000
Pythonistas	1000
Pythonistas	800
Rustaceans	800

Note how we have duplicate band names.

Hint

This is bad database design as we should add a unique constraint to prevent this, but go with it for this example!

Let's compare queries with and without `distinct`:

```
>>> await Band.count()
4
>>> await Band.count(distinct=[Band.name])
2
```

We can specify multiple columns:

```
>>> await Band.count(distinct=[Band.name, Band.popularity])
3
```

In the above example, this means we count rows where the combination of `name` and `popularity` is unique.

So ('Pythonistas', 1000) is a distinct value from ('Pythonistas', 800), because even though the `name` is the same, the `popularity` is different.

2.3.3 Clauses

where

See *where*.

2.4 Alter

This is used to modify an existing table.

Hint

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

2.4.1 add_column

Used to add a column to an existing table.

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

2.4.2 drop_column

Used to drop an existing column.

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

2.4.3 drop_table

Used to drop the table - use with caution!

```
await Band.alter().drop_table()
```

drop_db_tables / drop_db_tables_sync

If you have several tables which you want to drop, you can use *drop_db_tables* or *drop_db_tables_sync*. The tables will be dropped in the correct order based on their foreign keys.

```
# async version
>>> from piccolo.table import drop_db_tables
>>> await drop_db_tables(Band, Manager)

# sync version
>>> from piccolo.table import drop_db_tables_sync
>>> drop_db_tables_sync(Band, Manager)
```

2.4.4 rename_column

Used to rename an existing column.

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

2.4.5 set_null

Set whether a column is nullable or not.

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

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

2.4.6 set_schema

Used to change the `schema` which a table belongs to.

```
await Band.alter().set_schema('schema_1')
```

Schemas are a way of organising the tables within a database. Only Postgres and Cockroach support schemas. [Learn more here](#).

After changing a table's schema, you need to update your `Table` accordingly, otherwise subsequent queries will fail, as they'll be trying to find the table in the old schema.

```
Band._meta.schema = 'schema_1'
```

2.4.7 set_unique

Used to change whether a column is unique or not.

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

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

2.5 Create Table

This creates the table and columns in the database.

Hint

You can use migrations instead of manually altering the schema - see [Migrations](#).

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

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

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

2.5.1 create_db_tables / create_db_tables_sync

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:

```
# async version
>>> from piccolo.table import create_db_tables
>>> await create_db_tables(Band, Manager, if_not_exists=True)

# sync version
```

(continues on next page)

(continued from previous page)

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

2.6 Delete

This deletes any matching rows from the table.

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

2.6.1 force

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

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

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

2.6.2 Query clauses

returning

See *returning*.

where

See *where*.

2.7 Exists

This checks whether any rows exist which match the criteria.

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

2.7.1 Query clauses

where

See *where*.

2.8 Insert

This is used to bulk insert rows into the table:

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

2.8.1 add

If we later decide to insert additional rows, we can use the add method:

```
query = Band.insert(Band(name="Pythonistas"))

if other_bands:
    query = query.add(
        Band(name="Darts"),
        Band(name="Gophers")
    )

await query
```

2.8.2 Query clauses

on_conflict

See *on_conflict*.

returning

See *returning*.

2.9 Raw

Should you need to, you can execute raw SQL.

```
>>> await Band.raw('SELECT name FROM band')
[{'name': 'Pythonistas'}]
```

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

```
>>> await Band.raw('SELECT * FROM band WHERE name = {}'.format('Pythonistas'))
[{'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.

2.9.1 Query clauses

batch

See *batch*.

2.10 Update

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

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

2.10.1 force

Piccolo won't let you run an update query without a *where clause*, unless you explicitly tell it to do so. This is to prevent accidentally overwriting the data in a table.

```
>>> await Band.update()
Raises: UpdateError

# Works fine:
>>> await Band.update({Band.popularity: 0}, force=True)

# Or just add a where clause:
>>> await Band.update({Band.popularity: 0}).where(Band.popularity < 50)
```

2.10.2 Modifying values

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

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

Integer columns

You can add / subtract / multiply / divide values:

```
# Add 100 to the popularity of each band:
await Band.update(
    {
        Band.popularity: Band.popularity + 100
    },
    force=True
)
```

(continues on next page)

(continued from previous page)

```

# Decrease the popularity of each band by 100.
await Band.update(
  {
    Band.popularity: Band.popularity - 100
  },
  force=True
)

# Multiply the popularity of each band by 10.
await Band.update(
  {
    Band.popularity: Band.popularity * 10
  },
  force=True
)

# Divide the popularity of each band by 10.
await Band.update(
  {
    Band.popularity: Band.popularity / 10
  },
  force=True
)

# You can also use the operators in reverse:
await Band.update(
  {
    Band.popularity: 2000 - Band.popularity
  },
  force=True
)

```

Varchar / Text columns

You can concatenate values:

```

# Append "!!!" to each band name.
await Band.update(
  {
    Band.name: Band.name + "!!!"
  },
  force=True
)

# Concatenate the values in each column:
await Band.update(
  {
    Band.name: Band.name + Band.name
  },
  force=True
)

```

(continues on next page)

(continued from previous page)

```
# Prepend "!!!" to each band name.
await Band.update(
    {
        Band.popularity: "!!!" + Band.popularity
    },
    force=True
)
```

Date / Timestamp / Timestampz / Interval columns

You can add or subtract a `timedelta` to any of these columns.

For example, if we have a `Concert` table, and we want each concert to start one day later, we can simply do this:

```
await Concert.update(
    {
        Concert.starts: Concert.starts + datetime.timedelta(days=1)
    },
    force=True
)
```

Likewise, we can decrease the values by 1 day:

```
await Concert.update(
    {
        Concert.starts: Concert.starts - datetime.timedelta(days=1)
    },
    force=True
)
```

Array columns

You can append values to an array (Postgres only). See `cat`.

What about null values?

If we have a table with a nullable column:

```
class Band(Table):
    name = Varchar(null=True)
```

Any rows with a value of null aren't modified by an update:

```
>>> await Band.insert(Band(name="Pythonistas"), Band(name=None))
>>> await Band.update(
...     {
...         Band.name: Band.name + '!!!'
...     },
...     force=True
... )
>>> await Band.select()
# Note how the second row's name value is still `None`:
[{'id': 1, 'name': 'Pythonistas!!!'}, {'id': 2, 'name': None}]
```

It's more efficient to exclude any rows with a value of null using a *where clause*:

```
await Band.update(
    {
        Band.name + '!!!'
    },
    force=True
).where(
    Band.name.is_not_null()
)
```

2.10.3 Kwarg values

Rather than passing in a dictionary of values, you can use kwargs instead if you prefer:

```
await Band.update(
    name='Pythonistas 2'
).where(
    Band.name == 'Pythonistas'
)
```

2.10.4 Query clauses

returning

See *returning*.

where

See *where*.

2.11 Features

2.11.1 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.

Accessing the Engine

In the examples below we need to access the database Engine.

Each Table contains a reference to its Engine, which is the easiest way to access it. For example, with our Band table:

```
DB = Band._meta.db
```

Atomic

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

```
transaction = 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()
```

Transaction

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

```
async with DB.transaction():
    await Manager.create_table()
    await Concert.create_table()
```

Commit

The transaction is automatically committed when you exit the context manager.

```
async with DB.transaction():
    await query_1
    await query_2
    # Automatically committed if the code reaches here.
```

You can manually commit it if you prefer:

```
async with DB.transaction() as transaction:
    await query_1
    await query_2
    await transaction.commit()
    print('transaction committed!')
```

Rollback

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.

Rather than raising an exception, if you want to rollback a transaction manually you can do so as follows:

```
async with DB.transaction() as transaction:
    await Manager.create_table()
    await Band.create_table()
    await transaction.rollback()
```

Nested transactions

Nested transactions aren't supported in Postgres, but we can achieve something similar using `savepoints`.

Nested context managers

If you have nested context managers, for example:

```
async with DB.transaction():
    async with DB.transaction():
        ...
```

By default, the inner context manager does nothing, as we're already inside a transaction.

You can change this behaviour using `allow_nested=False`, in which case a `TransactionError` is raised if you try creating a transaction when one already exists.

```
async with DB.transaction():
    async with DB.transaction(allow_nested=False):
        # TransactionError('A transaction is already active.')
```

transaction_exists

You can check whether your code is currently inside a transaction using the following:

```
>>> DB.transaction_exists()
True
```

Savepoints

Postgres supports savepoints, which is a way of partially rolling back a transaction.

```
async with DB.transaction() as transaction:
    await Band.insert(Band(name='Pythonistas'))

    savepoint_1 = await transaction.savepoint()

    await Band.insert(Band(name='Terrible band'))

    # Oops, I made a mistake!
    await savepoint_1.rollback_to()
```

In the above example, the first query will be committed, but not the second.

Named savepoints

By default, we assign a name to the savepoint for you. But you can explicitly give it a name:

```
await transaction.savepoint('my_savepoint')
```

This means you can rollback to this savepoint at any point just using the name:

```
await transaction.rollback_to('my_savepoint')
```

Transaction types

SQLite

For SQLite you may want to specify the *transaction type*, as it can have an effect on how well the database handles concurrent requests.

2.11.2 Joins

Joins are handled automatically by Piccolo. They work everywhere you'd expect (select queries, where clauses, etc.).

A *fluent interface* is used, which lets you traverse foreign keys.

Here's an example of a select query which uses joins (using the *example schema*):

```
# This gets the band's name, and the manager's name by joining to the
# manager table:
>>> await Band.select(Band.name, Band.manager.name)
```

And a *where* clause which uses joins:

```
# This automatically joins with the manager table to perform the where
# clause. It only returns the columns from the band table though by default.
>>> await Band.select().where(Band.manager.name == 'Guido')
```

Left joins are used.

Improved static typing

You can optionally modify the above queries slightly for powerful static typing support from tools like Mypy and Pylance:

```
await Band.select(Band.name, Band.manager._.name)
```

Notice how we use `._` instead of `.` after each foreign key. An easy way to remember this is `._` looks a bit like a connector in a diagram.

Static type checkers now know that we're referencing the `name` column on the `Manager` table, which has many advantages:

- Autocompletion of column names.
- Easier code navigation (command + click on column names to navigate to the column definition).
- Most importantly, the detection of typos in column names.

This works, no matter how many joins are performed. For example:

```
await Concert.select(
    Concert.band_1._.name,
    Concert.band_1._.manager._.name,
)
```

Note

You may wonder why this syntax is required. We're operating within the limits of Python's typing support, which is still fairly young. In the future we will hopefully be able to offer identical static typing support for `Band.manager.name` and `Band.manager._.name`. But even then, the `._` syntax will still be supported.

join_on

Joins are usually performed using `ForeignKey` columns, though there may be situations where you want to join using a column which isn't a `ForeignKey`.

You can do this using `join_on`.

It's generally best to join on unique columns.

2.12 Comparisons

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

2.12.1 Django Comparison

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

Queries

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

Schema

OneToOneField

To do this in Piccolo, use a `ForeignKey` with a unique constraint - see *One to One*.

Database settings

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

Creating a new project

With Django you use `django-admin startproject mysite`.

In Piccolo you use `piccolo asgi new` (see *ASGI*).

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.

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

Likewise, with objects:

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

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

3.2 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.

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

Likewise, with objects:

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

3.3 order_by

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

- *Select*

- *Objects*

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

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

To order by descending:

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

You can specify the column name as a string if you prefer:

```
await Band.select().order_by(
    'name'
)
```

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

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

3.3.1 Advanced

Ascending and descending

If you want to order by multiple columns, with some ascending, and some descending, then you can do so using multiple `order_by` statements:

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

OrderByRaw

SQL's `ORDER BY` clause is surprisingly rich in functionality, and there may be situations where you want to specify the `ORDER BY` explicitly using SQL. To do this use `OrderByRaw`.

In the example below, we are ordering the results randomly:

```
from piccolo.query import OrderByRaw

await Band.select(Band.name).order_by(
    OrderByRaw('random()'),
)
```

The above is equivalent to the following SQL:

```
SELECT "band"."name" FROM band ORDER BY random() ASC
```

3.4 where

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

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

It allows powerful filtering of your data.

3.4.1 Equal / Not Equal

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

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

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)`

3.4.2 Greater than / less than

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

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

3.4.3 like / ilike

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

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

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

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

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

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

3.4.4 not_like

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

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

3.4.5 is_in / not_in

You can get all rows with a value contained in the list:

```
await Band.select().where(
    Band.name.is_in(['Pythonistas', 'Rustaceans'])
)
```

And all rows with a value not contained in the list:

```
await Band.select().where(
    Band.name.not_in(['Terrible Band', 'Awful Band'])
)
```

You can also pass a subquery into the is_in clause:

```
await Band.select().where(
    Band.id.is_in(
        Concert.select(Concert.band_1).where(
            Concert.starts >= datetime.datetime(year=2025, month=1, day=1)
        )
    )
)
```

(continues on next page)

(continued from previous page)

```
)
)
```

Hint

In SQL there are often several ways of solving the same problem. You can also solve the above using *join_on*.

```
>>> await Band.select().where(
...     Band.id.join_on(Concert.band_1).starts >= datetime.datetime(
...         year=2025, month=1, day=1
...     )
... )
```

Use whichever you prefer, and whichever suits the situation best.

Subqueries can also be passed into the `not_in` clause:

```
await Band.select().where(
    Band.id.not_in(
        Concert.select(Concert.band_1).where(
            Concert.starts >= datetime.datetime(year=2025, month=1, day=1)
        )
    )
)
```

3.4.6 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
await Band.select().where(
    Band.manager != None
)

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

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

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

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

3.4.7 Complex queries - and / or

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

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

await Band.select().where(
    (Band.popularity >= 100) | (Band.name == 'Pythonistas')
)
```

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)
```

Multiple where clauses

Using multiple where clauses is equivalent to an AND.

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

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

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

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

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

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

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

3.4.8 WhereRaw

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

```
from piccolo.columns.combination import WhereRaw

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

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"

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

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

```
from piccolo.columns.combination import WhereRaw

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

3.4.9 Joins

The where clause has full support for joins. For example:

```
>>> await Band.select(Band.name).where(Band.manager.name == 'Guido')
[{'name': 'Pythonistas'}]
```

3.4.10 Conditional where clauses

You can add where clauses conditionally (e.g. based on user input):

```
async def get_band_names(only_popular_bands: bool) -> list[str]:
    query = Band.select(Band.name).output(as_list=True)

    if only_popular_bands:
        query = query.where(Band.popularity >= 1000)

    return await query
```

Hint

This works with all clauses, not just where clauses.

3.5 batch

You can use batch clauses with the following queries:

- *Objects*
- *Raw*
- *Select*

3.5.1 Example

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)
```

3.5.2 Node

If you're using `extra_nodes` with `PostgresEngine`, you can specify which node to query:

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

3.5.3 Synchronous version

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())
```

3.6 callback

You can use callback clauses with the following queries:

- *Select*
- *Objects*

Callbacks are used to run arbitrary code after a query completes.

3.6.1 Callback handlers

A callback handler is a function or coroutine that takes query results as its only parameter.

For example, you can automatically print the result of a select query using `print` as a callback handler:

```
>>> await Band.select(Band.name).callback(print)
[{'name': 'Pythonistas'}]
```

Likewise for an objects query:

```
>>> await Band.objects().callback(print)
[<Band: 1>]
```

3.6.2 Transforming results

Callback handlers are able to modify the results of a query by returning a value. Note that in the previous examples, the queries returned `None` since `print` itself returns `None`.

To modify query results with a custom callback handler:

```
>>> def uppercase_name(band):
    return band.name.upper()

>>> await Band.objects().first().callback(uppercase_name)
'PYTHONISTAS'
```

3.6.3 Multiple callbacks

You can add as many callbacks to a query as you like. This can be done in two ways.

Passing a list of callbacks:

```
Band.select(Band.name).callback([handler_a, handler_b])
```

Chaining callback clauses:

```
Band.select(Band.name).callback(handler_a).callback(handler_b)
```

3.7 distinct

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

- *Select*

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

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

3.7.1 on

Using the `on` parameter we can create `DISTINCT ON` queries.

Note

Postgres and CockroachDB only. For more info, see the [Postgres docs](#).

If we have the following table:

```
class Album(Table):
    band = Varchar()
    title = Varchar()
    release_date = Date()
```

With this data in the database:

Table 1: Albums

id	band	title	release_date
1	Pythonistas	Py album 2021	2021-12-01
2	Pythonistas	Py album 2022	2022-12-01
3	Rustaceans	Rusty album 2021	2021-12-01
4	Rustaceans	Rusty album 2022	2022-12-01

To get the latest album for each band, we can do so with a query like this:

```
>>> await Album.select().distinct(
...     on=[Album.band]
... ).order_by(
...     Album.band
... ).order_by(
...     Album.release_date,
...     ascending=False
... )

[
  {
    'id': 2,
    'band': 'Pythonistas',
    'title': 'Py album 2022',
    'release_date': '2022-12-01'
  },
  {
    'id': 4,
    'band': 'Rustaceans',
    'title': 'Rusty album 2022',
    'release_date': '2022-12-01'
  },
]
```

The first column specified in `on` must match the first column specified in `order_by`, otherwise a *DistinctOnError* will be raised.

Source

class piccolo.query.mixins.**DistinctOnError**

Raised when DISTINCT ON queries are malformed.

3.8 freeze

You can use the freeze clause with any query type.

3.8.1 Source

Query.**freeze()** → 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
```

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.

3.9 group_by

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

- *Select*

It is used in combination with the *aggregate functions* - for example, **Count**.

3.9.1 Count

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

```
>>> from piccolo.query.functions.aggregate import Count

>>> await Band.select(
...     Band.manager.name.as_alias('manager_name'),
...     Count(alias='band_count')
... ).group_by(
...     Band.manager.name
... )

[
  {"manager_name": "Graydon", "band_count": 1},
  {"manager_name": "Guido", "band_count": 1}
]
```

3.9.2 Other aggregate functions

These work the same as `Count`. See *aggregate functions*.

3.10 having

You can use the `having` clause with the following queries:

- *Select*

It is used in combination with the *group_by* clause, to filter the grouped rows.

The syntax is identical to the *where* clause.

3.10.1 Example

In the following example, we get the number of albums per band (filtering out any bands with less than 2 albums).

Hint

You can run this query in the *Playground*.

```
>>> from piccolo.query.functions.aggregate import Count

>>> await Album.select(
...     Album.band.name.as_alias('band_name'),
...     Count()
... ).group_by(
...     Album.band
... ).having(
...     Count() >= 2
... )

[
  {"band_name": "Pythonistas", "count": 2},
]
```

3.11 lock_rows

You can use the `lock_rows` clause with the following queries:

- *Objects*
- *Select*

It returns a query that locks rows until the end of the transaction, generating a `SELECT ... FOR UPDATE` SQL statement or similar with other lock strengths.

Note

Postgres and CockroachDB only.

3.11.1 Basic Usage

Basic usage without parameters:

```
await Band.select(Band.name == 'Pythonistas').lock_rows()
```

Equivalent to:

```
SELECT ... FOR UPDATE
```

3.11.2 lock_strength

The parameter `lock_strength` controls the strength of the row lock when performing an operation in PostgreSQL. The value can be a predefined constant from the `LockStrength` enum or one of the following strings (case-insensitive):

- `UPDATE` (default): Acquires an exclusive lock on the selected rows, preventing other transactions from modifying or locking them until the current transaction is complete.
- `NO KEY UPDATE` (Postgres only): Similar to `UPDATE`, but allows other transactions to insert or delete rows that do not affect the primary key or unique constraints.
- `KEY SHARE` (Postgres only): Permits other transactions to acquire key-share or share locks, allowing non-key modifications while preventing updates or deletes.
- `SHARE`: Acquires a shared lock, allowing other transactions to read the rows but not modify or lock them.

You can specify a different lock strength:

```
await Band.select(Band.name == 'Pythonistas').lock_rows('SHARE')
```

Which is equivalent to:

```
SELECT ... FOR SHARE
```

3.11.3 nowait

If another transaction has already acquired a lock on one or more selected rows, an exception will be raised instead of waiting for the other transaction to release the lock.

```
await Band.select(Band.name == 'Pythonistas').lock_rows('UPDATE', nowait=True)
```

3.11.4 skip_locked

Ignore locked rows.

```
await Band.select(Band.name == 'Pythonistas').lock_rows('UPDATE', skip_locked=True)
```

3.11.5 of

By default, if there are many tables in a query (e.g. when joining), all tables will be locked. Using `of`, you can specify which tables should be locked.

```
await Band.select().where(Band.manager.name == 'Guido').lock_rows('UPDATE', of=(Band, ))
```

3.11.6 Full example

If we have this table:

```
class Concert(Table):
    name = Varchar()
    tickets_available = Integer()
```

And we want to make sure that `tickets_available` never goes below 0, we can do the following:

```
async def book_tickets(ticket_count: int):
    async with Concert._meta.db.transaction():
        concert = await Concert.objects().where(
            Concert.name == "Awesome Concert"
        ).first().lock_rows()

        if concert.tickets_available >= ticket_count:
            await concert.update_self({
                Concert.tickets_available: Concert.tickets_available - ticket_count
            })
        else:
            raise ValueError("Not enough tickets are available!")
```

This means that when multiple transactions are running at the same time, it isn't possible to book more tickets than are available.

i Note

There is a [video tutorial on YouTube](#).

3.11.7 Learn more

- [Postgres docs](#)
- [CockroachDB docs](#)

3.12 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.

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

Likewise, with objects:

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

3.13 on_conflict

Hint

This is an advanced topic, and first time learners of Piccolo can skip if they want.

You can use the `on_conflict` clause with the following queries:

- *Insert*

3.13.1 Introduction

When inserting rows into a table, if a unique constraint fails on one or more of the rows, then the insertion fails.

Using the `on_conflict` clause, we can instead tell the database to ignore the error (using `DO NOTHING`), or to update the row (using `DO UPDATE`).

This is sometimes called an **upsert** (update if it already exists else insert).

3.13.2 Example data

If we have the following table:

```
class Band(Table):
    name = Varchar(unique=True)
    popularity = Integer()
```

With this data:

id	name	popularity
1	Pythonistas	1000

Let's try inserting another row with the same name, and we'll get an error:

```
>>> await Band.insert(
...     Band(name="Pythonistas", popularity=1200)
... )
Unique constraint error!
```

3.13.3 DO NOTHING

To ignore the error:

```
>>> await Band.insert(
...     Band(name="Pythonistas", popularity=1200)
... ).on_conflict(
...     action="DO NOTHING"
... )
```

If we fetch the data from the database, we'll see that it hasn't changed:

```
>>> await Band.select().where(Band.name == "Pythonistas").first()
{'id': 1, 'name': 'Pythonistas', 'popularity': 1000}
```

3.13.4 DO UPDATE

Instead, if we want to update the popularity:

```
>>> await Band.insert(
...     Band(name="Pythonistas", popularity=1200)
... ).on_conflict(
...     action="DO UPDATE",
...     target=Band.name,
...     values=[Band.popularity]
... )
```

If we fetch the data from the database, we'll see that it was updated:

```
>>> await Band.select().where(Band.name == "Pythonistas").first()
{'id': 1, 'name': 'Pythonistas', 'popularity': 1200}
```

3.13.5 target

Using the `target` argument, we can specify which constraint we're concerned with. By specifying `target=Band.name` we're only concerned with the unique constraint for the `band` column. If you omit the `target` argument on `DO NOTHING` action, then it works for all constraints on the table. For `DO UPDATE` action, `target` is mandatory and must be provided.

```
>>> await Band.insert(
...     Band(name="Pythonistas", popularity=1200)
... ).on_conflict(
...     action="DO NOTHING",
...     target=Band.name
... )
```

If you want to target a composite unique constraint, you can do so by passing in a tuple of columns:

```
>>> await Band.insert(
...     Band(name="Pythonistas", popularity=1200)
... ).on_conflict(
...     action="DO NOTHING",
...     target=(Band.name, Band.popularity)
... )
```

You can also specify the name of a constraint using a string:

```
>>> await Band.insert(
...     Band(name="Pythonistas", popularity=1200)
... ).on_conflict(
...     action="DO NOTHING",
...     target='some_constraint'
... )
```

3.13.6 values

This lets us specify which values to update when a conflict occurs.

By specifying a *Column*, this means that the new value for that column will be used:

```
# The new popularity will be 1200.
>>> await Band.insert(
...     Band(name="Pythonistas", popularity=1200)
... ).on_conflict(
...     action="DO UPDATE",
...     values=[Band.popularity]
... )
```

Instead, we can specify a custom value using a tuple:

```
# The new popularity will be 1111.
>>> await Band.insert(
...     Band(name="Pythonistas", popularity=1200)
... ).on_conflict(
...     action="DO UPDATE",
...     values=(Band.popularity, 1111)
... )
```

If we want to update all of the values, we can use *all_columns*.

```
>>> await Band.insert(
...     Band(id=1, name="Pythonistas", popularity=1200)
... ).on_conflict(
```

(continues on next page)

(continued from previous page)

```
...     action="DO UPDATE",
...     values=Band.all_columns()
... )
```

3.13.7 where

This can be used with `DO UPDATE`. It gives us more control over whether the update should be made:

```
>>> await Band.insert(
...     Band(id=1, name="Pythonistas", popularity=1200)
... ).on_conflict(
...     action="DO UPDATE",
...     values=[Band.popularity],
...     where=Band.popularity < 1000
... )
```

3.13.8 Multiple on_conflict clauses

SQLite allows you to specify multiple `ON CONFLICT` clauses, but Postgres and Cockroach don't.

```
>>> await Band.insert(
...     Band(name="Pythonistas", popularity=1200)
... ).on_conflict(
...     action="DO UPDATE",
...     ...
... ).on_conflict(
...     action="DO NOTHING",
...     ...
... )
```

3.13.9 Learn more

- [Postgres docs](#)
- [Cockroach docs](#)
- [SQLite docs](#)

3.13.10 Source

```
Insert.on_conflict(target: str | Column | tuple[Column, ...] | None = None, action: OnConflictAction |
    Literal['DO NOTHING', 'DO UPDATE'] = OnConflictAction.do_nothing, values:
    Sequence[Column | tuple[Column, Any]] | None = None, where: Combinable | None =
    None) → Self
```

```
class piccolo.query.methods.insert.OnConflictAction(value)
```

Specify which action to take on conflict.

```
do_nothing = 'DO NOTHING'
```

```
do_update = 'DO UPDATE'
```

3.14 output

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

- *Select*
- *Objects*

3.14.1 Select queries only

as_json

To return the data as a JSON string:

```
>>> await Band.select(Band.name).output(as_json=True)
' [{"name": "Pythonistas"} ]'
```

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.

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

nested

Output any data from related tables in nested dictionaries.

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

3.14.2 Select and Objects queries

load_json

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

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

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

3.15 returning

You can use the returning clause with the following queries:

- *Insert*
- *Update*
- *Delete*

By default, an update query returns an empty list, but using the `returning` clause you can retrieve values from the updated rows.

```
>>> await Band.update({
...     Band.name: 'Pythonistas Tribute Band'
... }).where(
...     Band.name == 'Pythonistas'
... ).returning(Band.id, Band.name)
[{'id': 1, 'name': 'Pythonistas Tribute Band'}]
```

Similarly, for an insert query - we can retrieve some of the values from the inserted rows:

```
>>> await Manager.insert(
...     Manager(name="Maz"),
...     Manager(name="Graydon")
... ).returning(Manager.id, Manager.name)

[{'id': 1, 'name': 'Maz'}, {'id': 1, 'name': 'Graydon'}]
```

As another example, let's use delete and return the full row(s):

```
>>> await Band.delete().where(
...     Band.name == "Pythonistas"
... ).returning(*Band.all_columns())

[{'id': 1, 'name': 'Pythonistas', 'manager': 1, 'popularity': 1000}]
```

By counting the number of elements of the returned list, you can find out how many rows were affected or processed by the operation.

Warning

This works for all versions of Postgres, but only [SQLite 3.35.0](#) and above support the returning clause. See the [docs](#) on how to check your SQLite version.

3.16 as_of

Note

Cockroach only.

You can use `as_of` clause with the following queries:

- *Select*

- *Objects*

To retrieve historical data from 5 minutes ago:

```
await Band.select().where(  
    Band.name == 'Pythonistas'  
).as_of('-5min')
```

This generates an `AS OF SYSTEM TIME` clause. See [documentation](#).

This clause accepts a wide variety of time and interval [string formats](#).

This is very useful for performance, as it will reduce transaction contention across a cluster.

FUNCTIONS

Hint

This is an advanced topic - if you're new to Piccolo you can skip this for now.

Functions can be used to modify how queries are run, and what is returned.

4.1 Basic Usage

4.1.1 Select queries

Functions can be used in select queries - here's an example, where we convert the values to uppercase:

```
>>> from piccolo.query.functions import Upper

>>> await Band.select(
...     Upper(Band.name, alias="name")
... )

[{"name": "PYTHONISTAS"}]
```

4.1.2 Where clauses

Functions can also be used in where clauses.

```
>>> from piccolo.query.functions import Length

>>> await Band.select(
...     Band.name
... ).where(
...     Length(Band.name) > 10
... )

[{"name": "Pythonistas"}]
```

4.1.3 Update queries

And even in update queries:

```
>>> from piccolo.query.functions import Upper

>>> await Band.update(
...     {Band.name: Upper(Band.name)},
...     force=True
... ).returning(Band.name)

[{"name": "PYTHONISTAS"}, {"name": "RUSTACEANS"}, {"name": "C-SHARPS"}]
```

Pretty much everywhere.

4.2 Aggregate functions

4.2.1 Avg

```
class piccolo.query.functions.aggregate.Avg(identifier: Column | QueryString | str, alias: str | None = None)
```

AVG() SQL function. Column type must be numeric to run the query.

```
await Band.select(Avg(Band.popularity))

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

await Band.select(
    Avg(Band.popularity, alias="popularity_avg")
)

await Band.select(
    Avg(Band.popularity).as_alias("popularity_avg")
)
```

4.2.2 Count

```
class piccolo.query.functions.aggregate.Count(column: Column | None = None, distinct: Sequence[Column] | None = None, alias: str = 'count')
```

Used in Select queries, usually in conjunction with the `group_by` clause:

```
>>> await Band.select(
...     Band.manager.name.as_alias('manager_name'),
...     Count(alias='band_count')
... ).group_by(Band.manager)
[{'manager_name': 'Guido', 'count': 1}, ...]
```

It can also be used without the `group_by` clause (though you may prefer to the `Table.count` method instead, as it's more convenient):

```
>>> await Band.select(Count())
[{'count': 3}]
```

4.2.3 Min

`class piccolo.query.functions.aggregate.Min(identifier: Column | QueryString | str, alias: str | None = None)`

MIN() SQL function.

```
await Band.select(Min(Band.popularity))

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

await Band.select(
    Min(Band.popularity, alias="popularity_min")
)

await Band.select(
    Min(Band.popularity).as_alias("popularity_min")
)
```

4.2.4 Max

`class piccolo.query.functions.aggregate.Max(identifier: Column | QueryString | str, alias: str | None = None)`

MAX() SQL function.

```
await Band.select(
    Max(Band.popularity)
)

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

await Band.select(
    Max(Band.popularity, alias="popularity_max")
)

await Band.select(
    Max(Band.popularity).as_alias("popularity_max")
)
```

4.2.5 Sum

`class piccolo.query.functions.aggregate.Sum(identifier: Column | QueryString | str, alias: str | None = None)`

SUM() SQL function. Column type must be numeric to run the query.

```
await Band.select(
    Sum(Band.popularity)
)

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

await Band.select(
    Sum(Band.popularity, alias="popularity_sum")
)
```

(continues on next page)

```
)  
  
await Band.select(  
  Sum(Band.popularity).as_alias("popularity_sum")  
)
```

4.3 Array functions

4.3.1 ArrayCat

```
class piccolo.query.functions.array.ArrayCat(array_1: Column | QueryString | list[object], array_2:  
  Column | QueryString | list[object])
```

Concatenate two arrays.

Parameters

- **array_1** – These values will be at the start of the new array.
- **array_2** – These values will be at the end of the new array.

4.3.2 ArrayAppend

```
class piccolo.query.functions.array.ArrayAppend(array: Column | QueryString | list[object], value:  
  Column | QueryString | object)
```

Append an element to the end of an array.

Parameters

- **column** – Identifies the column.
- **value** – The value to append.

4.3.3 ArrayPrepend

```
class piccolo.query.functions.array.ArrayPrepend(array: Column | QueryString | list[object], value:  
  Column | QueryString | object)
```

Append an element to the beginning of an array.

Parameters

- **value** – The value to prepend.
- **column** – Identifies the column.

4.3.4 ArrayRemove

```
class piccolo.query.functions.array.ArrayRemove(array: Column | QueryString | list[object], value:  
  Column | QueryString | object)
```

Remove all elements equal to the given value from the array (array must be one-dimensional).

Parameters

- **column** – Identifies the column.
- **value** – The value to remove.

4.3.5 ArrayReplace

```
class piccolo.query.functions.array.ArrayReplace(array: Column | QueryString | list[object],
        old_value: Column | QueryString | object,
        new_value: Column | QueryString | object)
```

Replace each array element equal to the given value with a new value.

Parameters

- **column** – Identifies the column.
- **old_value** – The old value to be replaced.
- **new_value** – The new value we are replacing with.

4.4 Conditional functions

4.4.1 Coalesce

```
class piccolo.query.functions.conditional.Coalesce(*args: Column | QueryString | BasicTypes, alias:
        str | None = None)
```

Returns the first non-null value.

Here's an example to try in the playground:

```
>>> await Album.select(Album.release_date)
[
  {'release_date': datetime.date(2021, 1, 1)},
  {'release_date': datetime.date(2025, 1, 1)},
  {'release_date': datetime.date(2022, 2, 2)},
  {'release_date': None}
]
```

One of the values is null - we can specify a fallback value:

```
>>> from piccolo.functions.conditional import Coalesce
>>> await Album.select(
...     Coalesce(Album.release_date, datetime.date(2050, 1, 1))
... )
[
  {'release_date': datetime.date(2021, 1, 1)},
  {'release_date': datetime.date(2025, 1, 1)},
  {'release_date': datetime.date(2022, 2, 2)},
  {'release_date': datetime.date(2050, 1, 1)}
]
```

Or us this abbreviated syntax:

```
>>> await Album.select(
...     Album.release_date | datetime.date(2050, 1, 1)
... )
[
  {'release_date': datetime.date(2021, 1, 1)},
  {'release_date': datetime.date(2025, 1, 1)},
  {'release_date': datetime.date(2022, 2, 2)},
  {'release_date': datetime.date(2050, 1, 1)}
]
```

(continues on next page)

(continued from previous page)

```
{'release_date': datetime.date(2050, 1, 1)}
]
```

4.4.2 Nullif

`class piccolo.query.functions.conditional.NullIf(identifier: Column | QueryString, value: BasicTypes | QueryString, alias: str | None = None)`

Returns null if the value in the database equals value.

An example is where a Varchar or Text column contains a mix of empty strings and null. We might want to standardise the response so it's just null.

For example:

```
class Venue(Table):
    name = Varchar()
    address = Text(null=True)

>>> await Venue.select(Venue.name, NullIf(Venue.address, ''))
[{'name': 'Amazing venue', 'address': None}]
```

4.5 Datetime functions

4.5.1 Postgres / Cockroach

AtTimeZone

`class piccolo.query.functions.datetime.AtTimeZone(identifier: Time | Timestamp | Timestamptz | QueryString, timezone: ZoneInfo | str | timedelta, alias: str | None = None)`

Note

This is for Postgres / Cockroach only.

Convert the column to the given timezone. See the [Postgres docs](#) for more information.

For example:

```
class Signing(Table):
    starts = Timestamptz()

>>> await Signing.select(
...     AtTimeZone(Signing.starts, 'EST', alias='starts_est'),
...     Signing.starts,
... )
[{'starts_est': datetime.datetime(
    2026, 12, 20, 5, 0),
  'starts': datetime.datetime(
```

(continues on next page)

(continued from previous page)

```

    2026, 12, 20, 10, 0, tzinfo=datetime.timezone.utc
)
}]

```

Parameters

timezone – Valid arguments are 'EST', ZoneInfo('EST') and timedelta(hours=5).

Extract

```

class piccolo.query.functions.datetime.Extract(identifier: Date | Time | Timestamp | Timestamptz |
    QueryString, datetime_component: Literal['century',
    'day', 'decade', 'dow', 'doy', 'epoch', 'hour', 'isodow',
    'isoyear', 'julian', 'microseconds', 'millennium',
    'milliseconds', 'minute', 'month', 'quarter', 'second',
    'timezone', 'timezone_hour', 'timezone_minute', 'week',
    'year'], alias: str | None = None)

```

Note

This is for Postgres / Cockroach only.

Extract a date or time component from a Date / Time / Timestamp / Timestamptz column. For example, getting the month from a timestamp:

```

>>> from piccolo.query.functions import Extract
>>> await Concert.select(
...     Extract(Concert.starts, "month", alias="start_month")
... )
[{"start_month": 12}]

```

Parameters

- **identifier** – Identifies the column.
- **datetime_component** – The date or time component to extract from the column.

4.5.2 SQLite**Strftime**

```

class piccolo.query.functions.datetime.Strftime(identifier: Date | Time | Timestamp | Timestamptz |
    QueryString, datetime_format: str, alias: str | None =
    None)

```

Note

This is for SQLite only.

Format a datetime value. For example:

```
>>> from piccolo.query.functions import Strftime
>>> await Concert.select(
...     Strftime(Concert.starts, "%Y", alias="start_year")
... )
[{"start_month": "2024"}]
```

Parameters

- **identifier** – Identifies the column.
- **datetime_format** – A string describing the output format (see SQLite’s documentation for more info).

4.5.3 Database agnostic

These convenience functions work consistently across database engines.

They all work very similarly, for example:

```
>>> from piccolo.query.functions import Year
>>> await Concert.select(
...     Year(Concert.starts, alias="start_year")
... )
[{"start_year": 2024}]
```

Year

`piccolo.query.functions.datetime.Year(identifier: Date | Timestamp | Timestamptz | QueryString, alias: str | None = None) → QueryString`

Extract the year as an integer.

Month

`piccolo.query.functions.datetime.Month(identifier: Date | Timestamp | Timestamptz | QueryString, alias: str | None = None) → QueryString`

Extract the month as an integer.

Day

`piccolo.query.functions.datetime.Day(identifier: Date | Timestamp | Timestamptz | QueryString, alias: str | None = None) → QueryString`

Extract the day as an integer.

Hour

`piccolo.query.functions.datetime.Hour(identifier: Time | Timestamp | Timestamptz | QueryString, alias: str | None = None) → QueryString`

Extract the hour as an integer.

Minute

`piccolo.query.functions.datetime.Minute(identifier: Time | Timestamp | Timestamptz | QueryString, alias: str | None = None) → QueryString`

Extract the minute as an integer.

Second

`piccolo.query.functions.datetime.Second(identifier: Time | Timestamp | Timestamptz | QueryString, alias: str | None = None) → QueryString`

Extract the second as an integer.

4.6 Math functions

4.6.1 Abs

`class piccolo.query.functions.math.Abs(identifier: Column | QueryString | str, alias: str | None = None)`

Absolute value.

4.6.2 Ceil

`class piccolo.query.functions.math.Ceil(identifier: Column | QueryString | str, alias: str | None = None)`

Nearest integer greater than or equal to argument.

4.6.3 Floor

`class piccolo.query.functions.math.Floor(identifier: Column | QueryString | str, alias: str | None = None)`

Nearest integer less than or equal to argument.

4.6.4 Round

`class piccolo.query.functions.math.Round(identifier: Column | QueryString | str, alias: str | None = None)`

Rounds to nearest integer.

4.7 String functions

4.7.1 Concat

`class piccolo.query.functions.string.Concat(*args: Column | QueryString | str, alias: str | None = None)`

Concatenate multiple values into a single string.

Note

Null values are ignored, so `null + '!!!'` returns `!!!`, not `null`.

Warning

For SQLite, this is only available in version 3.44.0 and above.

4.7.2 Length

```
class piccolo.query.functions.string.Length(identifier: Column | QueryString | str, alias: str | None = None)
```

Returns the number of characters in the string.

4.7.3 Lower

```
class piccolo.query.functions.string.Lower(identifier: Column | QueryString | str, alias: str | None = None)
```

Converts the string to all lower case, according to the rules of the database's locale.

4.7.4 Ltrim

```
class piccolo.query.functions.string.Ltrim(identifier: Column | QueryString | str, alias: str | None = None)
```

Removes the longest string containing only characters in characters (a space by default) from the start of string.

4.7.5 Replace

```
class piccolo.query.functions.string.Replace(identifier: Column | QueryString, old: str, new: str, alias: str | None = None)
```

Replace any instances of *old* in the string with *new*.

For example, a really basic slugify implementation:

```
class Venue(Table):  
    name = Varchar()  
  
>>> await Venue.select(Replace(Venue.name, ' ', '-'))  
[{'name': 'Amazing-Venue'}]
```

4.7.6 Reverse

```
class piccolo.query.functions.string.Reverse(identifier: Column | QueryString | str, alias: str | None = None)
```

Return reversed string.

Not supported in SQLite.

4.7.7 Rtrim

```
class piccolo.query.functions.string.Rtrim(identifier: Column | QueryString | str, alias: str | None = None)
```

Removes the longest string containing only characters in characters (a space by default) from the end of string.

4.7.8 Upper

```
class piccolo.query.functions.string.Upper(identifier: Column | QueryString | str, alias: str | None = None)
```

Converts the string to all upper case, according to the rules of the database's locale.

4.8 Type conversion functions

4.8.1 Cast

```
class piccolo.query.functions.type_conversion.Cast(identifier: Column | QueryString | bytes | date |
datetime | time | timedelta | Decimal | dict | float |
int | list | str | UUID, as_type: Column, alias: str |
None = None)
```

Cast a value to a different type. For example:

```
>>> from piccolo.query.functions import Cast
>>> await Concert.select(
...     Cast(Concert.starts, Time(), alias="start_time")
... )
[{"start_time": datetime.time(19, 0)}]
```

You may also need Cast to explicitly tell the database which type you're sending in the query (though this is an edge case). Here is a contrived example:

```
>>> from piccolo.query.functions.math import Count
# This fails with asyncpg:
>>> await Band.select(Count([1,2,3]))
```

If we explicitly specify the type of the array, then it works:

```
>>> await Band.select(
...     Count(
...         Cast(
...             [1,2,3],
...             Array(Integer())
...         ),
...     )
... )
```

Parameters

- **identifier** – Identifies what is being converted (e.g. a column, or a raw value).
- **as_type** – The type to be converted to.

4.8.2 Notes on databases

Postgres and CockroachDB have very rich type systems, and you can convert between most types. SQLite is more limited.

The following query will work in Postgres / Cockroach, but you might get unexpected results in SQLite, because it doesn't have a native TIME column type:

```
>>> from piccolo.columns import Time
>>> from piccolo.query.functions import Cast
>>> await Concert.select(Cast(Concert.starts, Time()))
```

4.9 Custom functions

If there's a database function which Piccolo doesn't provide out of the box, you can still easily access it by using *QueryString* directly.

4.9.1 QueryString

QueryString is the building block of queries in Piccolo.

If we have a custom function defined in the database called `slugify`, you can access it like this:

```
from piccolo.querystring import QueryString

await Band.select(
    Band.name,
    QueryString('slugify({})', Band.name, alias='name_slug')
)
```

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

5.1 Defining a Schema

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

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.

5.1.1 Primary Key

Piccolo tables are automatically given a primary key column called `id`, which is an auto incrementing integer (a `Serial(primary_key=True)` column).

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)
```

5.1.2 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)
```

5.1.3 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*.

5.2 Column Types

Hint

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

5.2.1 Column

```
class piccolo.columns.base.Column(
    null: bool = False, primary_key: bool = False, unique: bool = False,
    index: bool = False, index_method: IndexMethod = IndexMethod.btree,
    required: bool = False, help_text: str | None = None, choices:
    type[Enum] | None = None, db_column_name: str | None = None,
    secret: bool = False, auto_update: Any = Ellipsis, **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 constraint 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")

>>> await MyTable.select(MyTable.class_)
[{'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)

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

- **auto_update** – Allows you to specify a value to set this column to each time it is updated (via `MyTable.update`, or `MyTable.save` on an existing row). A common use case is having a `modified_on` column.

```
class Band(Table):
    name = Varchar()
    popularity = Integer()
    # The value can be a function or static value:
    modified_on = Timestamp(auto_update=datetime.datetime.now)

# This will automatically set the `modified_on` column to the
# current timestamp, without having to explicitly set it:
>>> await Band.update({
...     Band.popularity: Band.popularity + 100
... }).where(Band.name == 'Pythonistas')
```

Note - this feature is implemented purely within the ORM. If you want similar functionality on the database level (i.e. if you plan on using raw SQL to perform updates), then you may be better off creating SQL triggers instead.

5.2.2 Bytea

```
class piccolo.columns.column_types.Bytea(default: bytes | bytearray | Enum | Callable[[], bytes] |
    Callable[[], bytearray] | None = b'', **kwargs:
    Unpack[ColumnKwargs])
```

Used for storing bytes.

Example

```

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

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

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

```

Hint

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

5.2.3 Boolean

```

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

```

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

Example

```

class Band(Table):
    has_drummer = Boolean()

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

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

```

5.2.4 ForeignKey

```

class piccolo.columns.column_types.ForeignKey(references: type[ReferencedTable], default: Any = None,
null: bool = True, on_delete: OnDelete =
OnDelete.cascade, on_update: OnUpdate =
OnUpdate.cascade, target_column: str | Column | None
= None, **kwargs)

```

```

class piccolo.columns.column_types.ForeignKey(references: LazyTableReference, default: Any = None,
null: bool = True, on_delete: OnDelete =
OnDelete.cascade, on_update: OnUpdate =
OnUpdate.cascade, target_column: str | Column | None
= None, **kwargs)

```

```

class piccolo.columns.column_types.ForeignKey(references: str, default: Any = None, null: bool = True,
on_delete: OnDelete = OnDelete.cascade, on_update:
OnUpdate = OnUpdate.cascade, target_column: 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
>>> await Band(manager=1).save()

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

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

Joins

You also use it to perform joins:

```
>>> await Band.select(Band.name, Band.manager.name).first()
{'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()
{'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
)
```

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

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

To change the manager:

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

Parameters

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

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

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

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

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

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

If the Table is defined in a Piccolo app:

```
from piccolo.columns.reference import LazyTableReference

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

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

```
from piccolo.columns.reference import LazyTableReference

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

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

Options:

- `OnDelete.cascade` (default)
- `OnDelete.restrict`
- `OnDelete.no_action`
- `OnDelete.set_null`
- `OnDelete.set_default`

To learn more about the different options, see the [Postgres docs](#).

```
from piccolo.columns import OnDelete
```

(continues on next page)

(continued from previous page)

```
class Band(Table):
    name = ForeignKey(
        references=Manager,
        on_delete=OnDelete.cascade
    )
```

- **on_update** – Determines what the database should do when a row has its primary key updated. If set to `OnUpdate.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 OnUpdate

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

5.2.5 Number

BigInt

```
class piccolo.columns.column_types.BigInt(default: int | Enum | Callable[[], int] | None = 0, **kwargs: Unpack[ColumnKwargs])
```

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
>>> await Band(popularity=1000000).save()

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

```

BigSerial

```

class piccolo.columns.column_types.BigSerial(null: bool = False, primary_key: bool = False, unique: bool = False, index: bool = False, index_method: IndexMethod = IndexMethod.btree, required: bool = False, help_text: str | None = None, choices: type[Enum] | None = None, db_column_name: str | None = None, secret: bool = False, auto_update: Any = Ellipsis, **kwargs)

```

An alias to a large autoincrementing integer column in Postgres.

Double Precision

```

class piccolo.columns.column_types.DoublePrecision(default: float | Enum | Callable[[], float] | None = 0.0, **kwargs: Unpack[ColumnKwargs])

```

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

Integer

```

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

```

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

Example

```

class Band(Table):
    popularity = Integer()

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

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

```

Numeric

```

class piccolo.columns.column_types.Numeric(digits: tuple[int, int] | None = None, default: Decimal | Enum | Callable[[], Decimal] | None = Decimal('0'), **kwargs: Unpack[ColumnKwargs])

```

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
>>> await Ticket(price=Decimal('50.0')).save()

# Query
>>> await Ticket.select(Ticket.price)
{'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: float | Enum | Callable[[], float] | None = 0.0,
**kwargs: Unpack[ColumnKwargs])

```

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
>>> await Concert(rating=7.8).save()

# Query
>>> await Concert.select(Concert.rating)
{'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: IndexMethod = IndexMethod.btree, required: bool = False, help_text: str | None = None, choices: type[Enum] | None = None, db_column_name: str | None = None, secret: bool = False, auto_update: Any = Ellipsis, **kwargs)
```

An alias to an autoincrementing integer column in Postgres.

SmallInt

```
class piccolo.columns.column_types.SmallInt(default: int | Enum | Callable[[], int] | None = 0, **kwargs: Unpack[ColumnKwargs])
```

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
>>> await Band(popularity=1000).save()

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

5.2.6 UUID

```
class piccolo.columns.column_types.UUID(default: UUID4 | UUID7 | UUID | str | Enum | None | Callable[[], UUID] = UUID4(), **kwargs: Unpack[ColumnKwargs])
```

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
>>> await DiscountCode(code=uuid.uuid4()).save()

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

5.2.7 Text

Char

```
class piccolo.columns.column_types.Char(length: int | None = 255, default: str | Enum | Callable[[], str] |
None = "", **kwargs: Unpack[ColumnKwargs])
```

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

Hint

`Varchar` is generally recommended over `Char`. However, `Char` may have slight performance advantages, and makes sense for certain fields like country codes where you know the values will be an exact length.

In Postgres the values are 'blank padded'. For example, if the length is 2, and you store 'X', then you will get 'X ' back from the database.

Note

In SQLite `Char` and `Varchar` behave identically.

Example

```
class Venue(Table):
    name = Varchar()
    country_code = Char(length=2)

# Create
>>> await Venue(name='Amazing Venue', country_code='GB').save()

# Query
>>> await Venue.select(Venue.country_code)
{'country_code': 'GB'}
```

Parameters

length – The maximum number of characters allowed.

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: str | Enum | None | Callable[[], str] = "", **kwargs:
Unpack[ColumnKwargs])
```

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()
```

(continues on next page)

(continued from previous page)

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

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

Varchar

```
class piccolo.columns.column_types.Varchar(length: int | None = 255, default: str | Enum | Callable[[], str] | None = "", **kwargs: Unpack[ColumnKwargs])
```

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
>>> await Band(name='Pythonistas').save()

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

Parameters

length – The maximum number of characters allowed.

Email

```
class piccolo.columns.column_types.Email(length: int | None = 255, default: str | Enum | Callable[[], str] | None = "", **kwargs: Unpack[ColumnKwargs])
```

Used for storing email addresses. It's identical to `Varchar`, except when using `create_pydantic_model` - we add email validation to the Pydantic model. This means that `Piccolo Admin` also validates email addresses.

5.2.8 Time

Date

```
class piccolo.columns.column_types.Date(default: DateOffset | DateCustom | DateNow | Enum | None | date | Callable[[], date] = DateNow(), **kwargs: Unpack[ColumnKwargs])
```

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

Example

```
import datetime

class Concert(Table):
    starts = Date()
```

(continues on next page)

(continued from previous page)

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

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

Interval

```
class piccolo.columns.column_types.Interval(default: IntervalCustom | Enum | None | timedelta |
Callable[[], timedelta] = IntervalCustom(weeks=0,
days=0, hours=0, minutes=0, seconds=0, milliseconds=0,
microseconds=0), **kwargs: Unpack[ColumnKwargs])
```

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

Example

```
from datetime import timedelta

class Concert(Table):
    duration = Interval()

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

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

Time

```
class piccolo.columns.column_types.Time(default: TimeCustom | TimeNow | TimeOffset | Enum | None |
time | Callable[[], time] = TimeNow(), **kwargs:
Unpack[ColumnKwargs])
```

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

Example

```
import datetime

class Concert(Table):
    starts = Time()

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

(continues on next page)

(continued from previous page)

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

Timestamp

```
class piccolo.columns.column_types.Timestamp(default: TimestampCustom | TimestampNow |
TimestampOffset | Enum | None | datetime |
DatetimeDefault | Callable[[], datetime] =
TimestampNow(), **kwargs: Unpack[ColumnKwargs])
```

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

Example

```
import datetime

class Concert(Table):
    starts = Timestamp()

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

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

Note

We recommend using `Timestamptz` columns if possible - having timezone information makes the timestamp unambiguous.

Timestamptz

```
class piccolo.columns.column_types.Timestamptz(default: TimestamptzCustom | TimestamptzNow |
TimestamptzOffset | Enum | None | datetime |
Callable[[], datetime] = TimestamptzNow(), **kwargs:
Unpack[ColumnKwargs])
```

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
>>> await Concert(
```

(continues on next page)

(continued from previous page)

```

...     starts=datetime.datetime(
...         year=2050, month=1, day=1, tzinfo=datetime.timezone.tz
...     )
... ).save()

# Query
>>> await Concert.select(Concert.starts)
{
  'starts': datetime.datetime(
    2050, 1, 1, 0, 0, tzinfo=datetime.timezone.utc
  )
}

```

5.2.9 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 / JSONB

```
class piccolo.columns.column_types.JSON(default: str | list | dict | Callable[[], str | list | dict] | None = '{}',
                                       **kwargs: Unpack[ColumnKwargs])
```

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.

```
class piccolo.columns.column_types.JSONB(default: str | list | dict | Callable[[], str | list | dict] | None =
                                         '{}', **kwargs: Unpack[ColumnKwargs])
```

Used for storing JSON strings - Postgres / CockroachDB only. The data is stored in a binary format, and can be queried more efficiently. 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.

Serialising

Piccolo automatically converts Python values into JSON strings:

```

studio = RecordingStudio(
    name="Abbey Road",
    facilities={"restaurant": True, "mixing_desk": True} # Automatically serialised
)
await studio.save()

```

You can also pass in a JSON string if you prefer:

```
studio = RecordingStudio(  
  name="Abbey Road",  
  facilities='{"restaurant": true, "mixing_desk": true}'  
)  
await studio.save()
```

Deserialising

The contents of a JSON / JSONB column are returned as a string by default:

```
>>> await RecordingStudio.select(RecordingStudio.facilities)  
[{"facilities": '{"restaurant": true, "mixing_desk": true}'}]
```

However, we can ask Piccolo to deserialise the JSON automatically (see *load_json*):

```
>>> await RecordingStudio.select(  
...   RecordingStudio.facilities  
... ).output(  
...   load_json=True  
... )  
[{"facilities": {"restaurant": True, "mixing_desk": True}}]
```

With objects queries, we can modify the returned JSON, and then save it:

```
studio = await RecordingStudio.objects().get(  
  RecordingStudio.name == 'Abbey Road'  
)  
.output(load_json=True)  
  
studio['facilities']['restaurant'] = False  
await studio.save()
```

Getting elements

JSON and JSONB columns have an arrow method (representing the \rightarrow operator in Postgres), which is useful for retrieving a child element from the JSON data.

Note

Postgres and CockroachDB only.

select queries

If we have the following JSON stored in the `RecordingStudio.facilities` column:

```
{  
  "instruments": {  
    "drum_kits": 2,  
    "electric_guitars": 10  
  },  
  "restaurant": true,  
  "technicians": [  
    {
```

(continues on next page)

(continued from previous page)

```

        "name": "Alice Jones"
    },
    {
        "name": "Bob Williams"
    }
]
}

```

We can retrieve the `restaurant` value from the JSON object:

```

>>> await RecordingStudio.select(
...     RecordingStudio.facilities.arrow('restaurant')
...     .as_alias('restaurant')
... ).output(load_json=True)
[{'restaurant': True}, ...]

```

As a convenience, you can use square brackets, instead of calling `arrow` explicitly:

```

>>> await RecordingStudio.select(
...     RecordingStudio.facilities['restaurant']
...     .as_alias('restaurant')
... ).output(load_json=True)
[{'restaurant': True}, ...]

```

You can drill multiple levels deep by calling `arrow` multiple times (or alternatively use the `from_path` method - see below).

Here we fetch the number of drum kits that the recording studio has:

```

>>> await RecordingStudio.select(
...     RecordingStudio.facilities["instruments"]["drum_kits"]
...     .as_alias("drum_kits")
... ).output(load_json=True)
[{'drum_kits': 2}, ...]

```

If you have a JSON object which consists of arrays and objects, then you can navigate the array elements by passing in an integer to `arrow`.

Here we fetch the first technician from the array:

```

>>> await RecordingStudio.select(
...     RecordingStudio.facilities["technicians"][0]["name"]
...     .as_alias("technician_name")
... ).output(load_json=True)
[{'technician_name': 'Alice Jones'}, ...]

```

where clauses

The arrow operator can also be used for filtering in a where clause:

```

>>> await RecordingStudio.select(RecordingStudio.name).where(
...     RecordingStudio.facilities['mixing_desk'].eq(True)

```

(continues on next page)

(continued from previous page)

```
... )
[{'name': 'Abbey Road'}]
```

from_path

This works the same as `arrow` but is more optimised if you need to return part of a highly nested JSON structure.

```
>>> await RecordingStudio.select(
...     RecordingStudio.facilities.from_path([
...         "technicians",
...         @,
...         "name"
...     ]).as_alias("technician_name")
... ).output(load_json=True)
[{'technician_name': 'Alice Jones'}, ...]
```

Handling null

When assigning a value of `None` to a JSON or JSONB column, this is treated as null in the database.

```
await RecordingStudio(name="ABC Studios", facilities=None).save()

>>> await RecordingStudio.select(
...     RecordingStudio.facilities
... ).where(
...     RecordingStudio.name == "ABC Studios"
... )
[{'facilities': None}]
```

If instead you want to store JSON null in the database, assign a value of `'null'` instead.

```
await RecordingStudio(name="ABC Studios", facilities='null').save()

>>> await RecordingStudio.select(
...     RecordingStudio.facilities
... ).where(
...     RecordingStudio.name == "ABC Studios"
... )
[{'facilities': 'null'}]
```

5.2.10 Array

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

```
class piccolo.columns.column_types.Array(base_column: Column, default: list | Enum | Callable[[], list] |
None = list, **kwargs: Unpack[ColumnKwargs])
```

Used for storing lists of data.

Example

```

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

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

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

```

Accessing individual elements

Array.**__getitem__**(value: int) → 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 (see [Postgres array docs](#)).

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

For example:

```

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

```

any

Array.**any**(value: Any) → Where

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

```

>>> await Ticket.select().where(Ticket.seat_numbers.any(510))

```

not_any

Array.**not_any**(value: Any) → Where

Check if the given value isn't in the array.

```

>>> await Ticket.select().where(Ticket.seat_numbers.not_any(510))

```

all

Array.**all**(value: Any) → Where

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

```

>>> await Ticket.select().where(Ticket.seat_numbers.all(510))

```

cat

Array.**cat**(value: ArrayType) → QueryString

A convenient way of accessing the [ArrayCat](#) function.

Used in an update query to concatenate two arrays.

```

>>> await Ticket.update({
...     Ticket.seat_numbers: Ticket.seat_numbers.cat([1000])
... }).where(Ticket.id == 1)

```

You can also use the + symbol if you prefer. To concatenate to the end:

```
>>> await Ticket.update({
...   Ticket.seat_numbers: Ticket.seat_numbers + [1000]
... }).where(Ticket.id == 1)
```

To concatenate to the start:

```
>>> await Ticket.update({
...   Ticket.seat_numbers: [1000] + Ticket.seat_numbers
... }).where(Ticket.id == 1)
```

You can concatenate multiple arrays in one go:

```
>>> await Ticket.update({
...   Ticket.seat_numbers: [1000] + Ticket.seat_numbers + [2000]
... }).where(Ticket.id == 1)
```

Note

Postgres / CockroachDB only

remove

Array.**remove**(value: ArrayItemType) → QueryString

A convenient way of accessing the [ArrayRemove](#) function.

Used in an update query to remove an item from an array.

```
>>> await Ticket.update({
...   Ticket.seat_numbers: Ticket.seat_numbers.remove(1000)
... }).where(Ticket.id == 1)
```

Note

Postgres / CockroachDB only

append

Array.**append**(value: ArrayItemType) → QueryString

A convenient way of accessing the [ArrayAppend](#) function.

Used in an update query to append an item to an array.

```
>>> await Ticket.update({
...   Ticket.seat_numbers: Ticket.seat_numbers.append(1000)
... }).where(Ticket.id == 1)
```

Note

Postgres / CockroachDB only

prepend

Array.**prepend**(value: ArrayItemType) → QueryString

A convenient way of accessing the [ArrayPrepend](#) function.

Used in an update query to prepend an item to an array.

```
>>> await Ticket.update({
...   Ticket.seat_numbers: Ticket.seat_numbers.prepend(1000)
... }).where(Ticket.id == 1)
```

Note

Postgres / CockroachDB only

replace

Array.**replace**(old_value: ArrayItemType, new_value: ArrayItemType) → QueryString

A convenient way of accessing the [ArrayReplace](#) function.

Used in an update query to replace each array item equal to the given value with a new value.

```
>>> await Ticket.update({
...   Ticket.seat_numbers: Ticket.seat_numbers.replace(1000, 500)
... }).where(Ticket.id == 1)
```

Note

Postgres / CockroachDB only

5.3 M2M

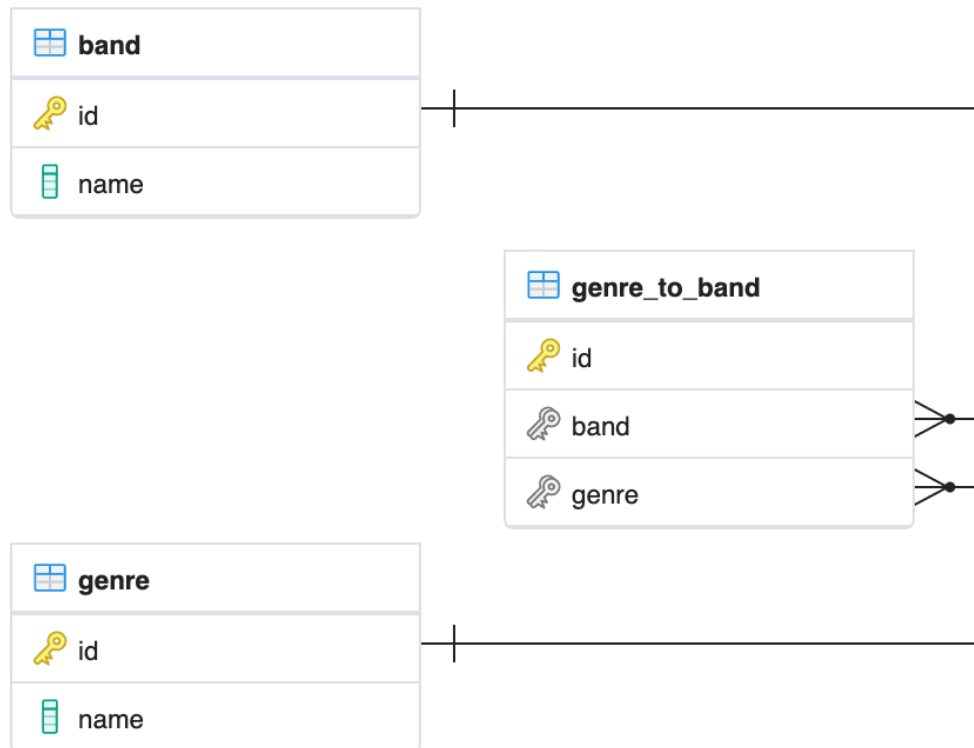
Note

There is a [video tutorial on YouTube](#).

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.

5.3.1 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'}
    ]
  }
]
```

(continues on next page)

(continued from previous page)

```

    },
    ...
]

```

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"]},
]

```

5.3.2 Objects queries

Piccolo makes it easy working with objects and M2M relationship.

add_m2m

`Table.add_m2m(*rows: Table, m2m: M2M, extra_column_values: dict[Column | str, Any] = {}) → 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
... )
[{'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: 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: Table, *m2m: 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()`.

5.4 One to One

5.4.1 Schema

A one to one relationship is basically just a foreign key with a unique constraint. In Piccolo, you can do it like this:

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

class Band(Table):
    name = Varchar()

class FanClub(Table):
    band = ForeignKey(Band, unique=True) # <- Note the unique constraint
    address = Text()
```

5.4.2 Queries

Getting a related object

If we have a Band object:

```
band = await Band.objects().where(Band.name == "Pythonistas").first()
```

To get the associated FanClub object, you could do this:

```
fan_club = await FanClub.objects().where(FanClub.band == band).first()
```

Or alternatively, using `get_related`:

```
fan_club = await band.get_related(Band.id.join_on(FanClub.band))
```

Instead of using `join_on`, you can use `reverse` to traverse the foreign key backwards if you prefer:

```
fan_club = await band.get_related(FanClub.band.reverse())
```

Select

If doing a select query, and you want data from the related table:

```
>>> await Band.select(
...     Band.name,
...     Band.id.join_on(FanClub.band).address.as_alias("address")
... )
[{'name': 'Pythonistas', 'address': '1 Flying Circus, UK'}, ...]
```

Where

If you want to filter by related tables in the `where` clause:

```
>>> await Band.select(
...     Band.name,
... ).where(Band.id.join_on(FanClub.band).address.like("%Flying%"))
[{'name': 'Pythonistas'}]
```

5.4.3 Source

`ForeignKey.reverse()` → *ForeignKey*

If there's a unique foreign key, this function reverses it.

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

class FanClub(Table):
    band = ForeignKey(Band, unique=True)
    address = Text()

class Treasurer(Table):
    fan_club = ForeignKey(FanClub, unique=True)
    name = Varchar()
```

It's helpful with `get_related`, for example:

```
>>> band = await Band.objects().first()
>>> await band.get_related(FanClub.band.reverse())
<Fan Club: 1>
```

It works multiple levels deep:

```
>>> await band.get_related(Treasurer.fan_club._.band.reverse())
<Treasurer: 1>
```

5.5 Advanced

5.5.1 Schemas

Postgres and CockroachDB have a concept called **schemas**.

It's a way of grouping the tables in a database. To learn more:

- [Postgres docs](#)
- [CockroachDB docs](#)

To specify a table's schema, do the following:

```
class Band(Table, schema="music"):
    ...

# The table will be created in the `music` schema.
# The music schema will also be created if it doesn't already exist.
>>> await Band.create_table()
```

If the schema argument isn't specified, then the table is created in the public schema.

Migration support

Schemas are fully supported in *database migrations*. For example, if we change the schema argument:

```
class Band(Table, schema="music_2"):
    ...
```

Then create an automatic migration and run it, then the table will be moved to the new schema:

```
>>> piccolo migrations new my_app --auto
>>> piccolo migrations forwards my_app
```

SchemaManager

The *SchemaManager* class is used internally by Piccolo to interact with schemas. You may find it useful if you want to write a script to interact with schemas (create / delete / list etc).

5.5.2 Readable

Sometimes Piccolo needs a succinct representation of a row - for example, when displaying a link in the *Piccolo Admin*. 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
```

(continues on next page)

(continued from previous page)

```
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.

```
await Band.select(Band.get_readable())
```

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.

5.5.3 Table Tags

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

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

5.5.4 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()
```

5.5.5 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.

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

>>> await Shirt.select()
[{'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.

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

>>> await Shirt.select().where(Shirt.size == Shirt.Size.small)
[{'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

Array columns

You can also use choices with `Array` columns.

```
class Ticket(Table):
    class Extras(str, enum.Enum):
        drink = "drink"
        snack = "snack"
        program = "program"
```

(continues on next page)

```
extras = Array(Varchar(), choices=Extras)
```

Note how you pass choices to Array, and not the base_column:

```
# CORRECT:
Array(Varchar(), choices=Extras)

# INCORRECT:
Array(Varchar(choices=Extras))
```

We can then use the Enum in our queries:

```
>>> await Ticket.insert(
...     Ticket(extras=[Extras.drink, Extras.snack]),
...     Ticket(extras=[Extras.program]),
... )
```

5.5.6 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:

```
>>> await Band.select()
[{'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`.

5.5.7 How to create custom column types

Sometimes, the column types shipped with Piccolo don't meet your requirements, and you will need to define your own column types.

Generally there are two ways to define your own column types:

- Create a subclass of an existing column type; or
- Directly subclass the `Column` class.

Try to use the first method whenever possible because it is more straightforward and can often save you some work. Otherwise, subclass `Column`.

Example

In this example, we create a column type called `MyColumn`, which is fundamentally an `Integer` type but has a custom attribute `custom_attr`:

```
from piccolo.columns import Integer

class MyColumn(Integer):
    def __init__(self, *args, custom_attr: str = '', **kwargs):
        self.custom_attr = custom_attr
        super().__init__(*args, **kwargs)

    @property
    def column_type(self):
        return 'INTEGER'
```

 **Hint**

It is **important** to specify the `column_type` property, which tells the database engine the **actual** storage type of the custom column.

Now we can use `MyColumn` in our table:

```
from piccolo.table import Table

class MyTable(Table):
    my_col = MyColumn(custom_attr='foo')
    ...
```

And later we can retrieve the value of the attribute:

```
>>> MyTable.my_col.custom_attr
'foo'
```

PROJECTS AND APPS

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

6.1 Piccolo Projects

A Piccolo project is a collection of apps.

6.1.1 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
```

6.1.2 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"]
)
```

6.1.3 DB

The DB setting is an Engine instance (see the *Engine docs*).

6.1.4 APP_REGISTRY

The APP_REGISTRY setting is an AppRegistry instance.

```
class piccolo.conf.apps.AppRegistry(apps: list[str] | None = None)
```

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']`.

6.2 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.
-

6.2.1 Creating an app

Run the following command within your project:

```
piccolo app new my_app --register
```

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`. If you used the `--register` flag, then this is done automatically. Otherwise, add it manually:

```
# 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*.

root

By default the app is created in the current directory. If you want the app to be in a sub folder, you can use the `--root` option:

```
piccolo app new my_app --register --root=./apps
```

The app will then be created in the `apps` folder.

6.2.2 AppConfig

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

```
# piccolo_app.py
import os

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

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

APP_CONFIG = AppConfig(
```

(continues on next page)

(continued from previous page)

```

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*.

You can register them manually (see the example above), or can use *table_finder*.

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.

6.2.3 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 modules list can contain absolute paths (e.g. `'blog.tables'`) or relative paths (e.g. `'tables'`). If relative paths are used, then the `package` argument must be passed in (`'blog'` in this case).

You can filter the `Table` subclasses returned using `tags`.

Source

```
piccolo.conf.apps.table_finder(modules: Sequence[str], package: str | None = None, include_tags:
    Sequence[str] | None = None, exclude_tags: Sequence[str] | None = None,
    exclude_imported: bool = False) → list[type[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'].
- **package** – This must be passed in if the modules are relative paths (e.g. if modules=['.tables'] then package='blog').
- **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)
```

6.2.4 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.

6.3 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.

6.3.1 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 built-in 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 types and versions.
- Fixtures are stored in JSON, which are a bit friendlier for source control.

dump

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
```

load

To load the fixture:

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

If you load the fixture again, you will get primary key errors because the rows already exist in the database. But what if we need to run it again, because we had a typo in our fixture, or were missing some data? We can upsert the data using `--on_conflict`.

There are two options:

1. `DO NOTHING` - if any of the rows already exist in the database, just leave them as they are, and don't raise an exception.
2. `DO UPDATE` - if any of the rows already exist in the database, override them with the latest data in the fixture file.

```
# DO NOTHING
piccolo fixtures load fixtures.json --on_conflict='DO NOTHING'

# DO UPDATE
piccolo fixtures load fixtures.json --on_conflict='DO UPDATE'
```

And finally, if you're loading a really large fixture, you can specify the `chunk_size`. By default, Piccolo inserts up to 1,000 rows at a time, as the database adapter will complain if a single insert query is too large. So if your fixture contains 10,000 rows, this will mean 10 insert queries.

You can tune this number higher or lower if you want (lower if the table has a lot of columns, or higher if the table has few columns).

```
piccolo fixtures load fixtures.json --chunk_size=500
```

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 its 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.

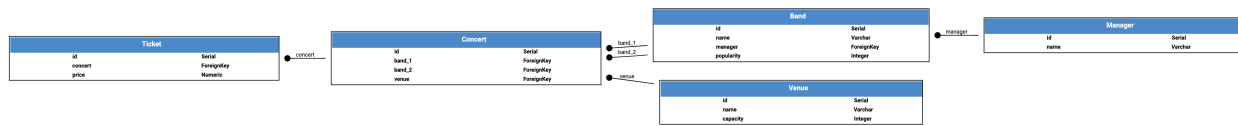
```
piccolo schema graph
```

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:



Note

There is a [video tutorial on YouTube](#).

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
```

Note

There is a [video tutorial on YouTube](#).

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).

Note

There is a [video tutorial on YouTube](#).

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"
```

6.3.2 Optional includes

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

user

Provides a user table, and commands for creating / managing users. See [Authentication](#).

Note

There is a [video tutorial on YouTube](#).

ENGINES

Engines are what execute the SQL queries. Each supported backend has its own *engine*.

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`.

7.1 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()
```

7.2 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.

7.2.1 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 entrypoint 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
```

7.3 Engine types

 **Hint**

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

7.3.1 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', log_queries: bool = False,
                                          log_responses: bool = False, **connection_kwargs)
```

Parameters

- **path** – A relative or absolute path to the the SQLite database file (it will be created if it doesn't already exist).
- **log_queries** – If True, all SQL and DDL statements are printed out before being run. Useful for debugging.
- **log_responses** – If True, the raw response from each query is printed out. Useful for debugging.
- **connection_kwargs** – These are passed directly to the database adapter. We recommend setting `timeout` if you expect your application to process a large number of concurrent writes, to prevent queries timing out. See Python's [sqlite3 docs](#) for more info.

Production tips

If you're planning on using SQLite in production with Piccolo, with lots of concurrent queries, then here are some *useful tips*.

7.3.2 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

See [Connection Pool](#).

Source

```
class piccolo.engine.postgres.PostgresEngine(config: dict[str, Any], extensions: Sequence[str] = (),
                                             log_queries: bool = False, log_responses: bool = False,
                                             extra_nodes: Mapping[str, PostgresEngine] | None =
                                             None)
```

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}`.

See the [asyncpg docs](#) for all available options.

- **extensions** – When the engine starts, it will try and create these extensions in Postgres. If you're using a read only database, set this value to an empty tuple `()`.
- **log_queries** – If True, all SQL and DDL statements are printed out before being run. Useful for debugging.
- **log_responses** – If True, the raw response from each query is printed out. Useful for debugging.
- **extra_nodes** – If you have additional database nodes (e.g. read replicas) for the server, you can specify them here. It's a mapping of a memorable name to a `PostgresEngine` instance. For example:

```
DB = PostgresEngine(
    config={'database': 'main_db'},
    extra_nodes={
        'read_replica_1': PostgresEngine(
            config={
                'database': 'main_db',
                host: 'read_replicate.my_db.com'
            },
```

(continues on next page)

(continued from previous page)

```

        extensions=()
    )
}
)

```

Note how we set `extensions=()`, because it's a read only database.

When executing a query, you can specify one of these nodes instead of the main database. For example:

```
>>> await MyTable.select().run(node="read_replica_1")
```

7.3.3 CockroachEngine

Configuration

```

# piccolo_conf.py
from piccolo.engine.cockroach import CockroachEngine

DB = CockroachEngine(config={
    'host': 'localhost',
    'database': 'piccolo',
    'user': 'root',
    'password': '',
    'port': '26257',
})

```

config

The config dictionary is passed directly to the underlying database adapter, `asyncpg`. See the [asyncpg docs](#) to learn more.

Connection Pool

See [Connection Pool](#).

Source

```

class piccolo.engine.cockroach.CockroachEngine(
    config: dict[str, Any], extensions: Sequence[str] = (),
    log_queries: bool = False, log_responses: bool =
    False, extra_nodes: dict[str, CockroachEngine] | None
    = None)

```

An extension of [PostgresEngine](#).

7.3.4 Connection Pool

 **Hint**

Connection pools can be used with Postgres and CockroachDB.

Setup

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. We also want to close the connection pool in the shutdown event handler.

The recommended way for Starlette and FastAPI apps is to use the `lifespan` parameter:

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

@asynccontextmanager
async def lifespan(app: Starlette):
    engine = engine_finder()
    assert engine
    await engine.start_connection_pool()
    yield
    await engine.close_connection_pool()

app = Starlette(lifespan=lifespan)
```

In older versions of Starlette and FastAPI, you may need event handlers instead:

```
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)
```


MIGRATIONS

8.1 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*.

You can either manually populate migrations, or allow Piccolo to do it for you automatically.

We recommend using *auto migrations* where possible, as it saves you time.

8.1.1 Manual migrations

First, let's create an empty migration:

```
piccolo migrations new my_app
```

This creates a new migration file in the migrations folder of the app. By default, the migration filename is the name of the app, followed by a timestamp, but you can rename it to anything you want:

```
piccolo_migrations/  
my_app_2022_12_06T13_58_23_024723.py
```

Note

We changed the naming convention for migration files in version **0.102.0** (previously they were like `2022-12-06T13-58-23-024723.py`). As mentioned, the name isn't important - change it to anything you want. The new format was chosen because a Python file should start with a letter by convention.

The contents of an empty migration file looks like this:

```
from piccolo.apps.migrations.auto.migration_manager import MigrationManager  
  
ID = "2022-02-26T17:38:44:758593"  
VERSION = "0.69.2" # The version of Piccolo used to create it  
DESCRIPTION = "Optional description"  
  
async def forwards():
```

(continues on next page)

(continued from previous page)

```

manager = MigrationManager(
    migration_id=ID,
    app_name="my_app",
    description=DESCRIPTION
)

def run():
    # Replace this with something useful:
    print(f"running {ID}")

manager.add_raw(run)
return manager

```

The ID is very important - it uniquely identifies the migration, and shouldn't be changed.

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

Running raw SQL

If you want to run raw SQL within your migration, you can do so as follows:

```

from piccolo.apps.migrations.auto.migration_manager import MigrationManager
from piccolo.table import Table

ID = "2025-07-28T09:51:54:296860"
VERSION = "1.27.1"
DESCRIPTION = "Updating each band's popularity"

# This is just a dummy table we use to execute raw SQL with:
class RawTable(Table):
    pass

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

    #####
    # This will get run when using `piccolo migrations forwards`:

    async def run():
        await RawTable.raw('UPDATE band SET popularity={}', 1000)

    manager.add_raw(run)

    #####
    # If we want to run some code when reversing the migration,

```

(continues on next page)

(continued from previous page)

```
# using `piccolo migrations backwards`:

async def run_backwards():
    await RawTable.raw('UPDATE band SET popularity={}', 0)

manager.add_raw_backwards(run_backwards)

#####
# We must always return the MigrationManager:

return manager
```

Hint

You can learn more about *raw queries* [here](#).

Using your Table classes

In the above example, we executed raw SQL, but what if we wanted to use the Table classes from our project instead? We have to be quite careful with this. Here's an example:

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

# We're importing a table from our project:
from music.tables import Band

ID = "2025-07-28T09:51:54:296860"
VERSION = "1.27.1"
DESCRIPTION = "Updating each band's popularity"

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

    async def run():
        await Band.update({Band.popularity: 1000}, force=True)

    manager.add_raw(run)
    return manager
```

We want our migrations to be repeatable - so if someone runs them a year from now, they will get the same results.

By directly importing our tables, we have the following risks:

- If the Band class is deleted from the codebase, it could break old migrations.
- If we modify the Band class, perhaps by removing columns, this could also break old migrations.

Try and make your migration files independent of other application code, so they're self contained and repeatable. Even though it goes against DRY, it's better to copy the relevant tables into your migration file:

```
from piccolo.apps.migrations.auto.migration_manager import MigrationManager
from piccolo.columns.column_types import Integer
from piccolo.table import Table

ID = "2025-07-28T09:51:54:296860"
VERSION = "1.27.1"
DESCRIPTION = "Updating each band's popularity"

# We defined the table within the file, rather than importing it.
class Band(Table):
    popularity = Integer()

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

    async def run():
        await Band.update({Band.popularity: 1000}, force=True)

    manager.add_raw(run)
    return manager
```

Another alternative is to use the `MigrationManager.get_table_from_snapshot` method to get a table from the migration history. This is very convenient, especially if the table is large, with many foreign keys.

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

ID = "2025-07-28T09:51:54:296860"
VERSION = "1.27.1"
DESCRIPTION = "Updating each band's popularity"

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

    async def run():
        # We get a table from the migration history.
        Band = await manager.get_table_from_snapshot(
            app_name="music", table_class_name="Band"
        )
```

(continues on next page)

(continued from previous page)

```
await Band.update({"popularity": 1000}, force=True)

manager.add_raw(run)

return manager
```

8.1.2 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.

8.1.3 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.

8.2 Running migrations

Hint

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

8.2.1 Forwards

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

```
piccolo migrations forwards my_app
```

Multiple apps

If you have multiple apps you can run them all using:

```
piccolo migrations forwards all
```

Migration table

When running the migrations, Piccolo will automatically create a database table called `migration` if it doesn't already exist. Each time a migration is successfully ran, a new row is added to this table.

Fake

We can 'fake' running a migration - we record that it ran in the database without actually running it.

There are two ways to do this - by passing in the `--fake` flag on the command line:

```
piccolo migrations forwards my_app 2022-09-04T19:44:09 --fake
```

Or by setting `fake=True` on the `MigrationManager` within the migration file.

```
async def forwards():
    manager = MigrationManager(
        migration_id=ID,
        app_name="app",
        description=DESCRIPTION,
        fake=True
    )
    ...
```

This is useful if we started from an existing database using `piccolo schema generate`, and the initial migration we generated is for tables which already exist, hence we fake run it.

8.2.2 Reversing migrations

To reverse the migration, run the following command, specifying the ID of a migration:

```
piccolo migrations backwards my_app 2022-09-04T19:44:09
```

Piccolo will then reverse the migrations for the given app, starting with the most recent migration, up to and including the migration with the specified ID.

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

8.2.3 Preview

To see the SQL queries of a migration without actually running them, use the `--preview` flag.

This works when running migrations forwards:

```
piccolo migrations forwards my_app --preview
```

Or backwards:

```
piccolo migrations backwards 2022-09-04T19:44:09 --preview
```

8.2.4 Checking migrations

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

```
piccolo migrations check
```

8.2.5 Source

These are the underlying Python functions which are called, so you can see all available options. These functions are converted into a CI using `targ`.

```
async piccolo.apps.migrations.commands.forwards.forwards(app_name: str, migration_id: str = 'all',
                                                         fake: bool = False, preview: bool =
                                                         False)
```

Runs any migrations which haven't been run yet.

Parameters

- **app_name** – The name of the app to migrate. Specify a value of 'all' to run migrations for all apps.
- **migration_id** – Migrations will be ran up to and including this migration_id. Specify a value of 'all' to run all of the migrations. Specify a value of '1' to just run the next migration.
- **fake** – If set, will record the migrations as being run without actually running them.
- **preview** – If true, don't actually run the migration, just print the SQL queries

```
async piccolo.apps.migrations.commands.backwards.backwards(app_name: str, migration_id: str = '1',
                                                           auto_agree: bool = False, clean: bool =
                                                           False, preview: bool = False)
```

Undo migrations up to a specific migration.

Parameters

- **app_name** – The app to reverse migrations for. Specify a value of 'all' to reverse migrations for all apps.

- **migration_id** – Migrations will be reversed up to and including this migration_id. Specify a value of 'all' to undo all of the migrations. Specify a value of '1' to undo the most recent migration.
- **auto_agree** – If true, automatically agree to any input prompts.
- **clean** – If true, the migration files which have been run backwards are deleted from the disk after completing.
- **preview** – If true, don't actually run the migration, just print the SQL queries.

async piccolo.apps.migrations.commands.check.**check**(*app_name: str = 'all'*)

Lists all migrations which have and haven't ran.

Parameters

app_name – The name of the app to check. Specify a value of 'all' to check the migrations for all apps.

AUTHENTICATION

Piccolo ships with authentication support out of the box.

9.1 Registering the app

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

9.2 Tables

9.2.1 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
```

 **Warning**

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

list

List existing users.

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* 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

create_user / create_user_sync

To create a new user:

```
# From within a coroutine:  
await BaseUser.create_user(username="bob", password="abc123", active=True)  
  
# When not in an event loop:  
BaseUser.create_user_sync(username="bob", password="abc123", active=True)
```

It saves the user in the database, and returns the created `BaseUser` instance.

 **Note**

It is preferable to use this rather than instantiating and saving `BaseUser` directly, as we add additional validation.

login / login_sync

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(user="bob", password="abc123")

# When not in an event loop:
BaseUser.update_password_sync(user="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.

The minimum password length allowed is 6 characters.

Extending BaseUser

If you want to extend `BaseUser` with additional fields, we recommend creating a `Profile` table with a `ForeignKey` to `BaseUser`, which can include any custom fields.

```

from piccolo.apps.user.tables import BaseUser
from piccolo.columns import ForeignKey, Text, Varchar
from piccolo.table import Table

class Profile(Table):
    custom_user = ForeignKey(BaseUser)
    phone_number = Varchar()
    bio = Text()

```

Alternatively, you can copy the entire `user` app into your project, and customise it to fit your needs.

Source

class piccolo.apps.user.tables.**BaseUser**(**kwargs)

Provides a basic user, with authentication support.

async classmethod **create_user**(username: str, password: str, **extra_params) → BaseUser

Creates a new user, and saves it in the database. It is recommended to use this rather than instantiating and saving BaseUser directly, as we add extra validation.

Raises

ValueError – If the username or password is invalid.

Returns

The created BaseUser instance.

classmethod **create_user_sync**(username: str, password: str, **extra_params) → BaseUser

A sync equivalent of `create_user()`.

async classmethod **login**(username: str, password: str) → int | None

Make sure the user exists and the password is valid. If so, the `last_login` value is updated in the database.

Returns

The id of the user if a match is found, otherwise None.

classmethod **login_sync**(username: str, password: str) → int | None

A sync equivalent of `login()`.

async classmethod **update_password**(user: str | int, password: str)

The password is the raw password string e.g. 'password123'. The user can be a user ID, or a username.

classmethod **update_password_sync**(user: str | int, password: str)

A sync equivalent of `update_password()`.

9.3 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.

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.

10.1 Routing frameworks

Starlette, FastAPI, BlackSheep, Litestar, Ravyn, Lilya, Quart, Falcon and Sanic are supported.

10.1.1 Which to use?

All are great choices. FastAPI is built on top of Starlette and Ravyn is built on top of Lilya, so they're very similar. FastAPI, BlackSheep, Litestar and Ravyn are great if you want to document a REST API, as they have built-in OpenAPI support.

10.2 Web servers

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

SERIALIZATION

Piccolo uses `Pydantic` internally to serialize and deserialize data.

Using `create_pydantic_model` you can easily create Pydantic models for your application.

11.1 `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:
band = await Band.objects().get(Band.name == 'Pythonistas')
model = BandModel(**band.to_dict())

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

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

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

11.1.1 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,))
```

11.1.2 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:
band = await Band.objects(Band.manager).get(Band.name == 'Pythonistas')
model = BandModel(**band.to_dict())

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

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

Note

There is a [video tutorial on YouTube](#).

11.1.3 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)
```

11.1.4 pydantic_config

Hint

We used to have a `pydantic_config_class` argument in Piccolo prior to v1, but it has been replaced with `pydantic_config` due to changes in Pydantic v2.

You can specify a Pydantic `ConfigDict` to use as the base for the Pydantic model's config (see [docs](#)).

For example, let's set the `extra` parameter to tell pydantic how to treat extra fields (that is, fields that would not otherwise be in the generated model). The allowed values are:

- `'ignore'` (default): silently ignore extra fields
- `'allow'`: accept the extra fields and assigns them to the model
- `'forbid'`: fail validation if extra fields are present

So if we want to disallow extra fields, we can do:

```
from pydantic.config import ConfigDict

config: ConfigDict = {
    "extra": "forbid"
}

model = create_pydantic_model(
    table=MyTable,
    pydantic_config=config
)
```

11.1.5 Required fields

You can specify which fields are required using the `required` argument of `Column`. For example:

```
class Band(Table):
    name = Varchar(required=True)

BandModel = create_pydantic_model(Band)

# Omitting the field raises an error:
>>> BandModel()
ValidationError - name field required
```

You can override this behaviour using the `all_optional` argument. An example use case is when you have a model which is used for filtering, then you'll want all fields to be optional.

```
class Band(Table):
    name = Varchar(required=True)

BandFilterModel = create_pydantic_model(
    Band,
    all_optional=True,
    model_name='BandFilterModel',
)

# This no longer raises an exception:
>>> BandModel()
```

11.1.6 Subclassing the model

If the generated model doesn't perfectly fit your needs, you can subclass it to add additional fields, and to override existing fields.

```
class Band(Table):
    name = Varchar(required=True)

BandModel = create_pydantic_model(Band)

class CustomBandModel(BandModel):
    genre: str

>>> CustomBandModel(name="Pythonistas", genre="Rock")
```

Or even simpler still:

```
class BandModel(create_pydantic_model(Band)):
    genre: str
```

11.1.7 Avoiding type warnings

Some linters will complain if you use variables in type annotations:

```
BandModel = create_pydantic_model(Band)
```

(continues on next page)

(continued from previous page)

```
def my_function(band: BandModel): # Variable not allowed in type expression!
    ...
```

The fix is really simple:

```
# We now have a class instead of a variable:
class BandModel(create_pydantic_model(Band)):
    ...

def my_function(band: BandModel):
    ...
```

11.1.8 Source

```
piccolo.utils.pydantic.create_pydantic_model(table: type[Table], nested: bool | tuple[ForeignKey, ...] =
False, exclude_columns: tuple[Column, ...] = (),
include_columns: tuple[Column, ...] = (),
include_default_columns: bool = False,
include_readable: bool = False, all_optional: bool =
False, model_name: str | None = None, deserialize_json:
bool = False, recursion_depth: int = 0,
max_recursion_depth: int = 5, pydantic_config:
ConfigDict | None = None, json_schema_extra: dict[str,
Any] | None = None) → type[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.
- **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.
- **pydantic_config** – Allows you to configure some of Pydantic’s behaviour. See the [Pydantic docs](#) for more info.
- **json_schema_extra** – 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.model_json_schema()
{..., "my_extra_field": "Hello"}
```

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.

11.2 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.

Piccolo provides a few tools to make testing easier.

12.1 Test runner

Piccolo ships with a handy command for running your unit tests using pytest. See the *tester app*.

You can put your test files anywhere you like, but a good place is in a `tests` folder within your Piccolo app. The test files should be named like `test_*.py` or `*_test.py` for pytest to recognise them.

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

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 objects with minimal attributes, leaving nullable fields empty:

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

12.3 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.

12.3.1 create_db_tables / drop_db_tables

Here we use `create_db_tables` and `drop_db_tables` to create and drop the tables.

Note

The sync equivalents are `create_db_tables_sync` and `drop_db_tables_sync`, if you need your tests to be synchronous for some reason.

```
from unittest import IsolatedAsyncioTestCase

from piccolo.table import create_db_tables, drop_db_tables
from piccolo.conf.apps import Finder

TABLES = Finder().get_table_classes()
```

(continues on next page)

(continued from previous page)

```

class TestApp(IsolatedAsyncioTestCase):
    async def setUp(self):
        await create_db_tables(*TABLES)

    async def tearDown(self):
        await drop_db_tables(*TABLES)

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

```

You can remove this boiler plate by using `AsyncTransactionTest`, which does this for you.

12.3.2 Run migrations

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

```

from unittest import IsolatedAsyncioTestCase

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

class TestApp(IsolatedAsyncioTestCase):
    async def setUp(self):
        await run_forwards("all")

    async def tearDown(self):
        await run_backwards("all", auto_agree=True)

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

```

12.4 Testing async code

There are a few options for testing async code using pytest.

12.4.1 run_sync

You can call any async code using Piccolo's `run_sync` utility:

```

from piccolo.utils.sync import run_sync

async def get_data():
    ...

def test_get_data():

```

(continues on next page)

(continued from previous page)

```
rows = run_sync(get_data())
assert len(rows) == 1
```

It's preferable to make your tests natively async though.

12.4.2 pytest-asyncio

If you prefer using pytest's function based tests, then take a look at [pytest-asyncio](#). Simply install it using `pip install pytest-asyncio`, then you can then write tests like this:

```
async def test_select():
    rows = await MyTable.select()
    assert len(rows) == 1
```

12.4.3 IsolatedAsyncioTestCase

If you prefer class based tests, and are using Python 3.8 or above, then have a look at [IsolatedAsyncioTestCase](#) from Python's standard library. You can then write tests like this:

```
from unittest import IsolatedAsyncioTestCase

class MyTest(IsolatedAsyncioTestCase):
    async def test_select(self):
        rows = await MyTable.select()
        assert len(rows) == 1
```

Also look at the [IsolatedAsyncioTestCase](#) subclasses which Piccolo provides (see [AsyncTransactionTest](#) and [AsyncTableTest](#) below).

12.5 TestCase subclasses

Piccolo ships with some `unittest.TestCase` subclasses which remove boilerplate code from tests.

```
class piccolo.testing.test_case.AsyncTransactionTest(methodName='runTest')
```

Wraps each test in a transaction, which is automatically rolled back when the test finishes.

Warning

Python 3.11 and above only.

If your test suite just contains [AsyncTransactionTest](#) tests, then you can setup your database tables once before your test suite runs. Any changes made to your tables by the tests will be rolled back automatically.

Here's an example:

```
from piccolo.testing.test_case import AsyncTransactionTest

class TestBandEndpoint(AsyncTransactionTest):
```

(continues on next page)

(continued from previous page)

```

async def test_band_response(self):
    """
    Make sure the endpoint returns a 200.
    """
    band = Band({Band.name: "Pythonistas"})
    await band.save()

    # Using an API testing client, like httpx:
    response = await client.get(f"/bands/{band.id}/")
    self.assertEqual(response.status_code, 200)

```

We add a Band to the database, but any subsequent tests won't see it, as the changes are rolled back automatically.

class piccolo.testing.test_case.**AsyncTableTest**(*methodName='runTest'*)

Used for tests where we need to create Piccolo tables - they will automatically be created and dropped.

For example:

```

class TestBand(AsyncTableTest):
    tables = [Band]

    async def test_band(self):
        ...

```

class piccolo.testing.test_case.**TableTest**(*methodName='runTest'*)

Identical to *AsyncTableTest*, except it only work for sync tests. Only use this if you can't make your tests async (perhaps you're on Python 3.7 where *IsolatedAsyncioTestCase* isn't available).

For example:

```

class TestBand(TableTest):
    tables = [Band]

    def test_band(self):
        ...

```


13.1 Types and Tab Completion

13.1.1 Type annotations

The Piccolo codebase uses type annotations extensively. This means it has great tab completion support in tools like iPython and VSCode.

It also means it works well with type checkers like Mypy.

To learn more about how Piccolo achieves this, read [this article about type annotations](#), and [this article about descriptors](#).

13.1.2 Troubleshooting

Here are some issues you may encounter when using Mypy, or another type checker.

id column doesn't exist

If you don't explicitly declare a column on your table with `primary_key=True`, Piccolo creates a `Serial` column for you called `id`.

In the following situation, the type checker might complain that `id` doesn't exist:

```
await Band.select(Band.id)
```

You can fix this as follows:

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

class Band(Table):
    id: Serial # Add an annotation
    name = Varchar()
```

13.2 Security

13.2.1 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.

13.3 Syntax

13.3.1 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.

13.3.2 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'
```

PLAYGROUND

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

14.1 Advanced Playground Usage

14.1.1 Postgres

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

Install Postgres

See *the docs on settings up 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
```

14.1.2 CockroachDB

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

Install CockroachDB

See the [installation guide](#) for your OS.

Create database

The playground is for testing and learning purposes only, so you can start a CockroachDB [single node](#) with the `insecure` flag (for non-production testing only) like this:

```
cockroach start-single-node --insecure
```

After that, in a new terminal window, you can create a database like this:

```
cockroach sql --insecure --execute="DROP DATABASE IF EXISTS piccolo_playground CASCADE;  
↵CREATE DATABASE piccolo_playground;"
```

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

```
user: "root"  
password: ""  
host: "localhost" # or 127.0.0.1  
database: "piccolo_playground"  
port: 26257
```

Connecting

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

```
piccolo playground run --engine=cockroach
```

14.1.3 iPython

The playground is built on top of iPython. We provide sensible defaults out of the box for syntax highlighting etc. However, to use your own custom iPython profile (located in `~/.ipython`), do the following:

```
piccolo playground run --ipython_profile
```

See the [iPython docs](#) for more information.

15.1 Piccolo API

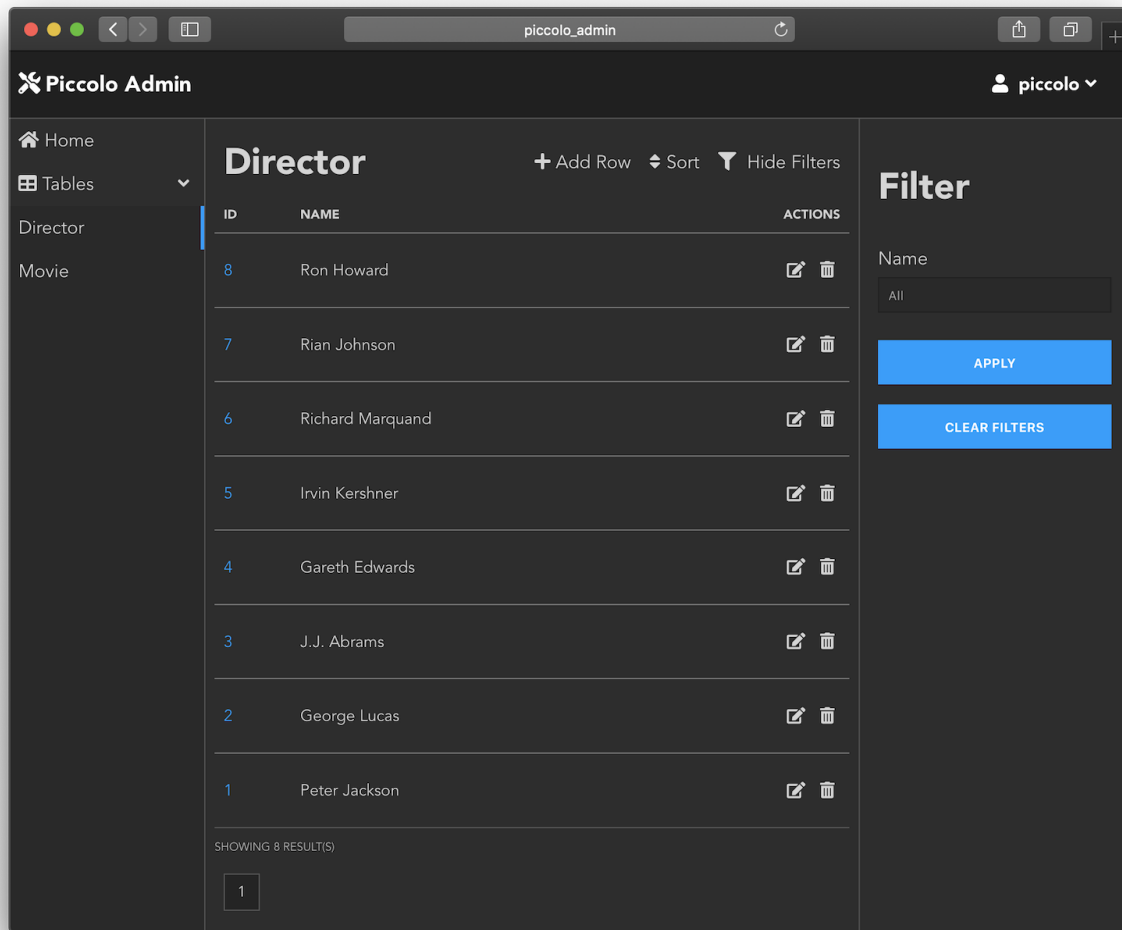
Provides some handy utilities for creating an API around your Piccolo tables. Examples include:

- 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](#), and the [docs](#) for more information.



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.

TUTORIALS

These tutorials bring together information from across the documentation, to help you solve common problems:

16.1 Migrate an existing project to Piccolo

16.1.1 Introduction

If you have an existing project and Postgres database, and you want to use Piccolo with it, these are the steps you need to take.

16.1.2 Option 1 - `piccolo asgi new`

This is the recommended way of creating brand new projects. If this is your first experience with Piccolo, then it's a good idea to create a test project:

```
mkdir test_project
cd test_project
piccolo asgi new
```

You'll learn a lot about how Piccolo works by looking at the generated code. You can then copy over the relevant files to your existing project if you like.

Alternatively, doing it from scratch, you'll need to do the following:

16.1.3 Option 2 - from scratch

Create a Piccolo project file

Create a new `piccolo_conf.py` file in the root of your project:

```
piccolo project new
```

This contains your database details, and is used to register Piccolo apps.

Create a new Piccolo app

The app contains your `Table` classes and migrations. Run this command at the root of your project:

```
# Replace 'my_app' with whatever you want to call your app
piccolo app new my_app
```

Register the new Piccolo app

Register this new app in `piccolo_conf.py`. For example:

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

While you're at it, make sure the database credentials are correct in `piccolo_conf.py`.

Make Table classes for your current database

Now, if you run:

```
piccolo schema generate
```

It will output Piccolo Table classes for your current database. Copy the output into `my_app/tables.py`. Double check that everything looks correct.

In `my_app/piccolo_app.py` make sure it's tracking these tables for migration purposes.

```
from piccolo.conf.apps import AppConfig, table_finder  
  
APP_CONFIG = AppConfig(  
    table_classes=table_finder(["my_app.tables"], exclude_imported=True),  
    ...  
)
```

Create an initial migration

This will create a new file in `my_app/piccolo_migrations`:

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

These tables already exist in the database, as it's an existing project, so you need to *fake apply* this initial migration:

```
piccolo migrations forwards my_app --fake
```

Making queries

Now you're basically setup - to make database queries:

```
from my_app.tables import MyTable  
  
async def my_endpoint():  
    data = await MyTable.select()  
    return data
```

Making new migrations

Just modify the files in `tables.py`, and then run:

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

16.2 Using SQLite and asyncio effectively

When using Piccolo with SQLite, there are some best practices to follow.

16.2.1 asyncio => lots of connections

With asyncio, we can potentially open lots of database connections, and attempt to perform concurrent database writes. SQLite doesn't support such concurrent behavior as effectively as Postgres, so we need to be careful.

One write at a time

SQLite can easily support lots of transactions concurrently if they are reading, but only one write can be performed at a time.

16.2.2 Transactions

SQLite has several transaction types, as specified by Piccolo's `TransactionType` enum:

```
class piccolo.engine.sqlite.TransactionType(value)
    See the SQLite docs for more info.
    deferred = 'DEFERRED'
    exclusive = 'EXCLUSIVE'
    immediate = 'IMMEDIATE'
```

Which to use?

When creating a transaction, Piccolo uses `DEFERRED` by default (to be consistent with SQLite).

This means that the first SQL query executed within the transaction determines whether it's a **READ** or **WRITE**:

- **READ** - if the first query is a `SELECT`
- **WRITE** - if the first query is something like an `INSERT / UPDATE / DELETE`

If a transaction starts off with a `SELECT`, but then tries to perform an `INSERT / UPDATE / DELETE`, SQLite tries to 'promote' the transaction so it can write.

The problem is, if multiple concurrent connections try doing this at the same time, SQLite will return a database locked error.

So if you're creating a transaction which you know will perform writes, then create an `IMMEDIATE` transaction:

```
from piccolo.engine.sqlite import TransactionType

async with Band._meta.db.transaction(
    transaction_type=TransactionType.immediate
):
    # We perform a SELECT first, but as it's an IMMEDIATE transaction,
    # we can later perform writes without getting a database locked
    # error.
    if not await Band.exists().where(Band.name == 'Pythonistas'):
        await Band.objects().create(name="Pythonistas")
```

Multiple IMMEDIATE transactions can exist concurrently - SQLite uses a lock to make sure only one of them writes at a time.

If your transaction will just be performing SELECT queries, then just use the default DEFERRED transactions - you will get improved performance, as no locking is involved:

```
async with Band._meta.db.transaction():
    bands = await Band.select()
    managers = await Manager.select()
```

16.2.3 timeout

It's recommended to specify the `timeout` argument in `SQLiteEngine`.

```
DB = SQLiteEngine(timeout=60)
```

Imagine you have a web app, and each endpoint creates a transaction which runs multiple queries. With SQLite, only a single write operation can happen at a time, so if several connections are open, they may be queued for a while.

By increasing `timeout` it means that queries are less likely to timeout.

To find out more about `timeout` see the Python `sqlite3` docs.

16.3 Deploying using Docker

16.3.1 Docker

Docker is a very popular way of deploying applications, using containers.

Base image

Piccolo has several dependencies which are compiled (e.g. `asynpg`, `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).

Environment variables

By using environment variables, we can inject the database credentials for Piccolo.

Example Dockerfile

This is a very simple Dockerfile, and illustrates the basics:

```
# Specify the base image:
FROM python:3.12-bookworm

# Install the pip requirements:
RUN pip install --upgrade pip
ADD app/requirements.txt /
```

(continues on next page)

(continued from previous page)

```
RUN pip install -r /requirements.txt

# Add the application code:
ADD app /app

# Environment variables:
ENV PG_HOST=localhost
ENV PG_PORT=5432
ENV PG_USER=my_database_user
ENV PG_PASSWORD=""
ENV PG_DATABASE=my_database

CMD ["/usr/local/bin/python", "/app/main.py"]
```

We can then modify our `piccolo_conf.py` file to use these environment variables:

```
# piccolo_conf.py

import os

DB = PostgresEngine(
    config={
        "port": int(os.environ.get("PG_PORT", "5432")),
        "user": os.environ.get("PG_USER", "my_database_user"),
        "password": os.environ.get("PG_PASSWORD", ""),
        "database": os.environ.get("PG_DATABASE", "my_database"),
        "host": os.environ.get("PG_HOST", "localhost"),
    }
)
```

When we run the container (usually via [Kubernetes](#), [Docker Compose](#), or similar), we can specify the database credentials using environment variables, which will be used by our application.

Accessing a local Postgres database

Bear in mind that if you have Postgres running locally on the server (i.e. on `localhost`), your Docker container won't automatically be able to access it. You can try Docker's host based networking, or just run Postgres within a Docker container.

16.4 FastAPI

FastAPI is a popular ASGI web framework. The purpose of this tutorial is to give some hints on how to get started with Piccolo and FastAPI.

Piccolo and FastAPI are a great match, and are commonly used together.

16.4.1 Creating a new project

Using the `piccolo asgi new` command, Piccolo will scaffold a new FastAPI app for you - simple!

16.4.2 Pydantic models

FastAPI uses *Pydantic* for serialising and deserialising data.

Piccolo provides *create_pydantic_model* which creates Pydantic models for you based on your Piccolo tables.

Of course, you can also just define your Pydantic models by hand.

16.4.3 Transactions

Using FastAPI's dependency injection system, we can easily wrap each endpoint in a transaction.

```

from fastapi import Depends, FastAPI
from pydantic import BaseModel

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

DB = SQLiteEngine()

class Band(Table, db=DB):
    """
    You would usually import this from tables.py
    """

    name = Varchar()

async def transaction():
    async with DB.transaction() as transaction:
        yield transaction

app = FastAPI()

@app.get("/bands/", dependencies=[Depends(transaction)])
async def get_bands():
    return await Band.select()

class CreateBandModel(BaseModel):
    name: str

@app.post("/bands/", dependencies=[Depends(transaction)])
async def create_band(model: CreateBandModel):
    await Band({Band.name: model.name}).save()

    # If an exception is raised then the transaction is rolled back.
    raise Exception("Oops")

```

(continues on next page)

(continued from previous page)

```

async def main():
    await Band.create_table(if_not_exists=True)

if __name__ == "__main__":
    import asyncio

    import uvicorn

    asyncio.run(main())
    uvicorn.run(app)

```

FastAPI dependencies can be declared at the endpoint, `APIRouter`, or even app level.

16.4.4 FastAPIWrapper

Piccolo API has a powerful utility called `FastAPIWrapper` which generates REST endpoints based on your Piccolo tables, and adds them to FastAPI's Swagger docs. It's a very productive way of building an API.

16.4.5 Authentication

Piccolo API ships with `authentication middleware` which is compatible with `FastAPI middleware`.

16.5 Avoiding circular imports

16.5.1 How Python imports work

When Python imports a file, it evaluates it from top to bottom.

With `ForeignKey` columns we sometimes have to reference tables lower down in the file (which haven't been evaluated yet).

The solutions are:

- Try and move the referenced table to a different Python file.
- Use `LazyTableReference`

16.5.2 Import Table definitions as early as possible

In the entrypoint to your app, at the top of the file, it's recommended to import your tables.

```

# main.py
from my_app.tables import Manager, Band

```

This ensures that the tables are imported, and setup correctly.

16.5.3 Keep table files focused

You should try and keep your `tables.py` files pretty focused (i.e. just contain your `Table` definitions).

If you have lots of logic alongside your `Table` definitions, it might cause your `LazyTableReference` references to evaluate too soon (causing circular import errors). An example of this is with `create_pydantic_model`:

```
# tables.py

from piccolo.columns import ForeignKey, Varchar
from piccolo.table import Table
from piccolo.utils.pydantic import create_pydantic_model

class Band(Table):
    name = Varchar()
    # This automatically gets converted into a LazyTableReference, because a
    # string is passed in:
    manager = ForeignKey("Manager")

# This is not recommended, as it will cause the LazyTableReference to be
# evaluated before Manager has imported.
# Instead, move this to a separate file, or below Manager.
BandModel = create_pydantic_model(Band)

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

16.5.4 Simplify your schema if possible

Even with *LazyTableReference*, you may run into some problems if your schema is really complicated.

An example is when you have two tables, and they have foreign keys to each other.

```
class Band(Table):
    name = Varchar()
    manager = ForeignKey("Manager")

class Manager(Table):
    name = Varchar()
    favourite_band = ForeignKey(Band)
```

Piccolo should be able to create these tables, and query them. However, some Piccolo tooling may struggle - for example when loading *fixtures*.

A joining table can help in these situations:

```
class Band(Table):
    name = Varchar()
    manager = ForeignKey("Manager")

class Manager(Table):
    name = Varchar()

class ManagerFavouriteBand(Table):
```

(continues on next page)

(continued from previous page)

```
manager = ForeignKey(Manager, unique=True)
band = ForeignKey(Band)
```

16.6 Moving a Table Between Piccolo Apps Without Data Loss

Piccolo ORM makes it easy to manage models within individual apps. But what if you need to move a table (model) from one app to another—say, from `app_a` to `app_b`—without losing your data?

This tutorial walks you through the safest way to move a table between Piccolo apps using migrations and the `--fake` flag.

16.6.1 Use Case

You're working on a project structured with multiple Piccolo apps, and you want to reorganize your models by moving a table (`TableA`) from one app (`app_a`) to another (`app_b`), without affecting the data in your database.

16.6.2 Prerequisites

- Piccolo ORM installed and configured
- Both `app_a` and `app_b` listed in `piccolo_conf.py` under `PICCOLO_APPS`
- Basic familiarity with Piccolo migrations

16.6.3 Step-by-Step Instructions

1. Remove the Table from `app_a`

In `app_a/tables.py`, delete or comment out the `TableA` class definition.

2. Create a Migration in `app_a`

Run the following command in your terminal:

```
piccolo migrations new app_a --auto
```

This will create a migration that removes the table from `app_a`.

3. Fake Apply the Migration

To prevent the table from actually being dropped from the database, apply the migration using the `--fake` flag:

```
piccolo migrations forwards app_a --fake
```

This marks the migration as applied without making real changes to the database.

4. Move the Table to `app_b`

Copy the `TableA` class definition into `app_b/tables.py`.

Ensure the definition matches exactly what it was in `app_a`.

5. Create a Migration in app_b

Generate a new fake migration for app_b to register TableA:

```
piccolo migrations new app_b --auto
```

6. Apply the Migration in app_b

Apply the new migration:

```
piccolo migrations forwards app_b --fake
```

Because the table already exists in the database, Piccolo will associate it with app_b without duplicating or altering it.

16.6.4 Notes & Tips

- This process preserves your data because it avoids actually dropping or creating the table.
- Always back up your database before doing schema changes.
- Inspect the migration files to understand what Piccolo is tracking.

16.7 UUID columns in Piccolo >= 1.31.0

Prior to piccolo==1.31.0, Postgres *UUID* columns generated their default values using `uuid_generate_v4()`, which is part of the `uuid-oss` extension.

In all the versions of Postgres that Piccolo supports, there is now a `gen_random_uuid()` function built in, which Piccolo now uses for all new UUID columns for generating default values in the database.

Your existing UUID columns will be left alone, and will continue to use the `uuid-oss` extension.

Note that *PostgresEngine* has an `extensions` argument. These are Postgres extensions which Piccolo will try to enable when starting up. This used to include `uuid-oss` by default, but this is no longer the case. If you want to maintain the old behaviour, simply do this:

```
DB = PostgresEngine(extensions=['uuid-oss'])
```

If you want to migrate an existing UUID column over to use `gen_random_uuid()` then run this script:

```
ALTER TABLE my_table ALTER COLUMN my_uuid_column SET DEFAULT gen_random_uuid();
```

CONTRIBUTING

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

17.1 Running Cockroach

To get a local Cockroach instance running, you can use:

```
cockroach start-single-node --insecure --store=type=mem,size=2GiB
```

Make sure the test database exists:

```
cockroach sql --insecure
>>> create database piccolo;
>>> use piccolo;
```

17.2 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`
 - Install database drivers: `pip install -r requirements/extras/postgres.txt -r requirements/extras/sqlite.txt`
 - Setup Postgres, and make sure a database called `piccolo` exists (see `tests/postgres_conf.py`).
 - 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 Cockroach: `./scripts/test-cockroach.sh`
 - Run the test suite with Sqlite: `./scripts/test-sqlite.sh`
-

17.3 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: `./scripts/run-docs.sh`
 - The docs will auto rebuild as you make changes.
-

17.4 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.

17.5 Profiling

This isn't required to contribute to Piccolo, but is useful when investigating performance problems.

- Install the dependencies: `pip install requirements/profile-requirements.txt`
- Make sure a Postgres database called `piccolo_profile` exists.
- Run `./scripts/profile.sh` to get performance data.

18.1 1.34.0

18.1.1 NullIf

Return null if a certain condition is met - for example, convert any empty strings to null:

```
from piccolo.query.functions import NullIf

class Venue(Table):
    address = Text()

>>> await Venue.select(NullIf(Venue.address, ''))
[{'address': None}]
```

18.1.2 Replace

String replacements within queries - for example, a simple slugify implementation:

```
from piccolo.query.functions import Replace

class Venue(Table):
    name = Varchar()

>>> await Venue.select(Replace(Venue.name, ' ', '-'))
[{'name': 'Amazing-Venue'}]
```

18.1.3 AtTimeZone

Convert Timestamptz columns from UTC to another timezone:

```
from piccolo.query.functions import AtTimeZone

class Signing(Table):
    starts = Timestamptz()

>>> await Signing.select(
...     AtTimeZone(Signing.starts, 'EST'),
... )
[{'starts': datetime.datetime(2026, 12, 20, 5, 0)}]
```

18.1.4 Char column

Similar to Varchar - useful if the strings are a fixed width.

It's also useful if using Piccolo on an existing database.

18.1.5 Fixture ordering

When dumping fixtures using `piccolo fixtures dump`, the data is now sorted by primary key, so subsequent dumps results in smaller Git diffs.

18.1.6 load_json with prefetch

When using `.output(load_json=True)` alongside `prefetch`, the child objects now load JSON strings into Python objects. Thanks to @diklios5768 for reporting this issue.

```
class RecordingStudio(Table):
    facilities = JSONB()

class Album(Table):
    recorded_at = ForeignKey(RecordingStudio)

>>> album = await Album.objects(Album.recorded_at).output(load_json=True).first()
>>> album.recorded_at.facilities
{
    'restaurant': True,
    'mixing_desk': True,
    'instruments': {'electric_guitars': 10, 'drum_kits': 2},
    'technicians': [{'name': 'Alice Jones'}, {'name': 'Bob Williams'}]
}
```

18.2 1.33.0

18.2.1 UUID v7

Added support for UUID v7.

```
from piccolo.columns.defaults.uuid import UUID7

class MyTable(Table):
    my_column = UUID(default=UUID7())
```

This provides superior insert performance and easier indexing compared to UUID v4.

Requires Python 3.14, and Postgres 18 (otherwise extensions are needed).

18.2.2 Coalesce

Added the Coalesce function - which lets you specify a fall back if a null value is found.

For example:

```
>>> await Album.select(
...     Coalesce(Album.release_date, datetime.date(2050, 1, 1))
... )
```

Or you can use this abbreviated syntax:

```
>>> await Album.select(
...     Album.release_date | datetime.date(2050, 1, 1)
... )
```

18.2.3 Other changes

- Fixed a bug with the Piccolo CLI, where custom names for commands weren't being applied (thanks to @sinisaos for this and @pelid for reporting the issue).
- Fixed typo in the `get_related` docstring (thanks to @nightcityblade for this).
- Fixed bugs with queries when a column has `db_column_name` defined (thanks to @VladislavYar for raising these issues).
- Improved the docs for `Timestamp` and `Timestamptz` columns.

18.3 1.32.0

Added the `having` clause, which is useful when working with `group_by`.

For example, here we get the number of albums per band, but exclude any bands with less than 2 albums:

```
>>> from piccolo.query.functions.aggregate import Count

>>> await Album.select(
...     Album.band.name.as_alias('band_name'),
...     Count()
... ).group_by(
...     Album.band
... ).having(
...     Count() >= 2
... )

[
  {"band_name": "Pythonistas", "count": 2},
]
```

We also updated our CockroachDB support to the latest version (thanks to @sinisaos for this).

18.4 1.31.0

- Added official Postgres 18 support.
- Removed graphlib backport (no longer needed in supported Python versions).
- Improved documentation for column kwargs.
- Updated README to mention other parts of the Piccolo ecosystem (thanks to @sinisaos for this).
- Table instances can now be compared using the `==` operator, and will return `True` if they have the same primary key value (thanks to @aarcex3 and @Skelmis for this).

- Fixed missing import for auto migrations when tables use `LazyTableReference` (thanks to @sinisaos for this).
 - Improved docs for the auto generated primary key (thanks to @badlydrawnrob for this).
 - Improved `Table._table_str` (this is most obvious in the playground, where the table definitions printed out now show more information - thanks to @badlydrawnrob for raising this issue).
 - Fixed a bug with `ModelBuilder` when using `Array` columns with choices defined.
 - `add_m2m` now returns the id of the joining table row, to match the docs (thanks to @diklios5768 for reporting this).
 - Improved the docs for `UUID` columns (what the `UUID4` default value does).
 - Moved connection pooling to its own page in the docs.
 - Improved the `on_conflict` clause - a `target` must be specified if using `DO UPDATE` (thanks to @mafuyuuu1 for raising this issue, and @sinisaos for this fix).
 - Changed Esmerald to Ravyn in the ASGI templates (thanks to @sinisaos for this).
 - Fixed a bug with auto migrations when changing a column to a foreign key - an exception was being raised (thanks to @gsavchuk for raising this issue and @sinisaos for this fix).
 - `Array(Numeric())` columns now work with `SQLite` (thanks to @sinisaos for this).
 - `UUID` columns now use the built-in `gen_random_uuid()` function in Postgres to generate a default value, instead of using the `uuid-oss` extension (thanks to @sinisaos for this). See [this tutorial](#) for more details.
-

18.5 1.30.0

Added support for Python 3.14 (thanks to @sinisaos for this).

18.6 1.29.0

- Fixed a bug with adding / subtracting `Integer` columns from one another in queries (thanks to @ryanvarley for this).
 - Updated the ASGI templates, and `BlackSheep` dependencies (thanks to @sinisaos for this).
 - Fixed a bug where decimal values generated by `ModelBuilder` could be too large.
 - Added an example `M2M` relationship in the playground to make learning `M2M` easier (thanks to @sinisaos for this).
 - Added documentation for `MigrationManager.get_table_from_snapshot`, which is a way of getting a `Table` from the migration history - useful when running data migrations (thanks to @sinisaos for this).
 - Columns with the `secret=True` argument are now added to `Table._meta.secret_columns` (thanks to @sinisaos for this).
 - Added documentation for the migration table.
 - Tidied up `Pydantic` tests (thanks to @sinisaos for this).
-

18.7 1.28.0

18.7.1 Playground improvements

- Added an Array column to the playground (Album.awards), for easier experimentation with array columns.
- CoachroachDB is now supported in the playground (thanks to @sinisaos for this).

```
piccolo playground run --engine=cockroach
```

18.7.2 Functions

Added lots of useful array functions (thanks to @sinisaos for this).

Here's an example, where we can easily fix a typo in an array using replace:

```
>>> await Album.update({
...     Album.awards: Album.awards.replace('Grammy Award 2021', 'Grammy Award 2022')
... }, force=True)
```

The documentation for functions has also been improved (e.g. how to create a custom function).

The Cast function is now more flexible.

18.7.3 Array concatenation

Values can be prepended:

```
>>> await Album.update({
...     Album.awards: ['Grammy Award 2020'] + Album.awards
... }, force=True)
```

And multiple arrays can be concatenated in one go:

```
>>> await Album.update({
...     Album.awards: ['Grammy Award 2020'] + Album.awards + ['Grammy Award 2025']
... }, force=True)
```

18.7.4 is_in and not_in sub queries

You can now use sub queries within is_in and not_in Thanks to @sinisaos for this.

```
>>> await Band.select().where(
...     Band.id.is_in(
...         Concert.select(Concert.band_1).where(
...             Concert.starts >= datetime.datetime(year=2025, month=1, day=1)
...         )
...     )
... )
```

18.7.5 Other improvements

- Auto convert a default value of 0 to 0.0 in Float columns.
- Modernised the type hints throughout the codebase (e.g. using list instead of typing.List). Thanks to @sinisaos for this.

- Fixed a bug with auto migrations, where the `Array` base column class wasn't being imported.
 - Improved M2M query performance by using sub selects (thanks to @sinisaos for this).
-

18.8 1.27.1

Improved the type annotations in `ColumnKwargs` - made some optional. Thanks to @stronk7 and @sinisaos for their help with this.

18.9 1.27.0

Improved auto completion / typo detection for column arguments.

For example:

```
class Band(Table):
    name = Varchar(nul=True) # linters will now warn that nul is a typo (should be null)
```

Thanks to @sinisaos for this.

18.10 1.26.1

Updated the BlackSheep ASGI template - thanks to @sinisaos for this.

Fixed a bug with auto migrations when a `ForeignKey` specifies `target_column` - multiple primary key columns were added to the migration file. Thanks to @waldner for reporting this issue.

Added a tutorial for moving tables between Piccolo apps - thanks to @sarvesh4396 for this.

18.11 1.26.0

Improved auto migrations - `ON DELETE` and `ON UPDATE` can be modified on `ForeignKey` columns. Thanks to @sinisaos for this.

18.12 1.25.0

Improvements to Piccolo app creation. When running the following:

```
piccolo app new my_app
```

It now validates that the app name (`my_app` in this case) is valid as a Python package.

Also, there is now a `--register` flag, which automatically adds the new app to the `APP_REGISTRY` in `piccolo_conf.py`.

```
piccolo app new my_app --register
```

Other changes:

- `table_finder` can now use relative modules.
- Updated the Esmerald ASGI template (thanks to @sinisaos for this).
- When using the `remove` method to delete a row from the database (`await some_band.remove()`), `some_band._exists_in_db` is now set to `False`. Thanks to @sinisaos for this fix.

18.13 1.24.2

Fixed a bug with `delete` queries which had joins in the `where` clause. For example:

```
>>> await Band.delete().where(Band.manager.name == 'Guido')
```

Thanks to @HakierGrzonzo for reporting the issue, and @sinisaos for the fix.

18.14 1.24.1

Fixed a bug with default values in `Timestamp` and `Timestamptz` columns. Thanks to @splch for this.

18.15 1.24.0

- Fixed a bug with `get_or_create` when a table has a column with both `null=False` and `default=None` - thanks to @bymoye for reporting this issue.
- If a `PostgresEngine` uses the `dsn` argument for `asyncpg`, this is now used by `piccolo sql_shell run`. Thanks to @abhishek-compro for suggesting this.
- Fixed the type annotation for the `length` argument of `Varchar` - it is allowed to be `None`. Thanks to @Compro-Prasad for this.

18.16 1.23.0

- Added Quart, Sanic, and Falcon as supported ASGI frameworks (thanks to @sinisaos for this).
- Fixed a bug with very large integers in SQLite.
- Fixed type annotation for `Timestamptz` default values (thanks to @Skelmis for this).

18.17 1.22.0

Python 3.13 is now officially supported.

JSON / JSONB querying has been significantly improved. For example, if we have this table:

```
class RecordingStudio(Table):
    facilities = JSONB()
```

And the `facilities` column contains the following JSON data:

```
{
  "technicians": [
    {"name": "Alice Jones"},
    {"name": "Bob Williams"},
  ]
}
```

We can get the first technician name as follows:

```
>>> await RecordingStudio.select(
...     RecordingStudio.facilities["technicians"][0]["name"].as_alias("name")
... ).output(load_json=True)
[{'name': 'Alice Jones'}, ...]
```

`TableStorage` (used for dynamically creating Piccolo Table classes from an existing database) was improved, to support a Dockerised version of Piccolo Admin, which is coming soon.

18.18 1.21.0

Postgres 17 is now officially supported.

Fixed a bug with joins, when a `ForeignKey` column had `db_column_name` specified. Thanks to @jessemcl-flwls for reporting this issue.

18.19 1.20.0

`get_related` now works multiple layers deep:

```
concert = await Concert.objects().first()
manager = await concert.get_related(Concert.band_1._.manager)
```

18.20 1.19.1

Fixed a bug with the `get_m2m` method, which would raise a `ValueError` when no objects were found. It now handles this gracefully and returns an empty list instead. Thanks to @nVitus for this fix.

Improved the ASGI templates (including a fix for the latest Litestar version). Thanks to @sinisaos for this.

18.21 1.19.0

Added support for row locking (i.e. `SELECT ... FOR UPDATE`).

For example, if we have this table:

```
class Concert(Table):
    name = Varchar()
    tickets_available = Integer()
```

And we want to make sure that `tickets_available` never goes below 0, we can do the following:

```
async def book_tickets(ticket_count: int):
    async with Concert._meta.db.transaction():
        concert = await Concert.objects().where(
            Concert.name == "Awesome Concert"
        ).first().lock_rows()

        if concert.tickets_available >= ticket_count:
            await concert.update_self({
                Concert.tickets_available: Concert.tickets_available - ticket_count
            })
        else:
            raise ValueError("Not enough tickets are available!")
```

This means that when multiple transactions are running at the same time, it isn't possible to book more tickets than are available.

Thanks to @dkopitsa for adding this feature.

18.22 1.18.0

18.22.1 update_self

Added the `update_self` method, which is an alternative to the `save` method. Here's an example where it's useful:

```
# If we have a band object:
>>> band = await Band.objects().get(name="Pythonistas")
>>> band.popularity
1000

# We can increment the popularity, based on the current value in the
# database:
>>> await band.update_self({
...     Band.popularity: Band.popularity + 1
... })

# The new value is set on the object:
>>> band.popularity
1001

# It's safer than using the `save` method, because the popularity value on
# the object might be out of date with what's in the database:
```

(continues on next page)

(continued from previous page)

```
band.popularity += 1
await band.save()
```

Thanks to @trondhindenes for suggesting this feature.

18.22.2 Batch raw queries

The batch method can now be used with raw queries. For example:

```
async with await MyTable.raw("SELECT * FROM my_table").batch() as batch:
    async for _batch in batch:
        print(_batch)
```

This is useful when you expect a raw query to return a lot of data.

Thanks to @devsarvesh92 for suggesting this feature.

18.23 1.17.1

Fixed a bug with migrations, where altering a column type from Integer to Float could fail. Thanks to @kurtportelli for reporting this issue.

18.24 1.17.0

Each migration is automatically wrapped in a transaction - this can now be disabled using the `wrap_in_transaction` argument:

```
manager = MigrationManager(
    wrap_in_transaction=False,
    ...
)
```

This is useful when writing a manual migration, and you want to manage all of the transaction logic yourself (or want multiple transactions).

granian is now a supported server in the ASGI templates. Thanks to @sinisaos for this.

18.25 1.16.0

Added custom async TestCase subclasses, to help with testing.

For example AsyncTransactionTest, which wraps each test in a transaction automatically:

```
class TestBandEndpoint(AsyncTransactionTest):

    async def test_band_response(self):
        """
```

(continues on next page)

(continued from previous page)

```

Make sure the endpoint returns a 200.
"""
# This data automatically gets removed from the database when the
# test finishes:
band = Band({Band.name: "Pythonistas"})
await band.save()

# Using an API testing client, like httpx:
response = await client.get(f"/bands/{band.id}/")
self.assertEqual(response.status_code, 200)

```

And AsyncTableTest, which automatically creates and drops tables:

```

class TestBand(AsyncTableTest):

    # These tables automatically get created and dropped:
    tables = [Band]

    async def test_band(self):
        ...

```

18.26 1.15.0

Improved refresh - it now works with prefetched objects. For example:

```

>>> band = await Band.objects(Band.manager).first()
>>> band.manager.name
"Guido"

# If the manager has changed in the database, when we refresh the band, the
# manager object will also be updated:
>>> await band.refresh()
>>> band.manager.name
"New name"

```

Also, improved the error messages when creating a BaseUser - thanks to @haavk for this.

18.27 1.14.0

Laying the foundations for alternative Postgres database drivers (e.g. `psqlpy`). Thanks to @insani7y and @chandr-andr for their help with this.

Warning

The SQL generated by Piccolo changed slightly in this release. Aliases used to be like "manager\$name" but now they are like "manager.name" (note \$ changed to .). If you are using `SelectRaw` in your queries to refer to these columns, then they will need updating. Please let us know if you encounter any other issues.

18.28 1.13.1

In Piccolo 1.6.0 we moved some aggregate functions to a new file. We now re-export them from their original location to keep backwards compatibility. Thanks to @sarvesh-deserve for reporting this issue.

18.29 1.13.0

Improved LazyTableReference, to help prevent circular import errors.

18.30 1.12.0

- Added documentation for one to one fields.
 - Upgraded ASGI templates (thanks to @sinisaos for this).
 - Migrations can now be hardcoded as fake.
 - Refactored tests to reduce boilerplate code.
 - Updated documentation dependencies.
-

18.31 1.11.0

Added datetime functions, for example Year:

```
>>> from piccolo.query.functions import Year
>>> await Concert.select(Year(Concert.starts, alias="starts_year"))
[{'starts_year': 2024}]
```

Added the Concat function, for concatenating strings:

```
>>> from piccolo.query.functions import Concat
>>> await Band.select(
...     Concat(
...         Band.name,
...         '-',
...         Band.manager._.name,
...         alias="name_and_manager"
...     )
... )
[{"name_and_manager": "Pythonistas-Guido"}]
```

18.32 1.10.0

Added not_any method for Array columns. This will return rows where an array doesn't contain the given value. For example:

```

class MyTable(Table):
    array_column = Array(Integer())

>>> await MyTable.select(
...     MyTable.array_column
... ).where(
...     MyTable.array_column.not_any(1)
... )
[{"array_column": [4, 5, 6]}]

```

Also fixed a bunch of Pylance linter warnings across the codebase.

18.33 1.9.0

Added some math functions, for example Abs, Ceil, Floor and Round.

```

>>> from piccolo.query.functions import Round
>>> await Ticket.select(Round(Ticket.price, alias="price"))
[{'price': 50.0}]

```

Added more operators to QueryString (multiply, divide, modulus, power), so we can do things like:

```

>>> await Ticket.select(Round(Ticket.price) * 2)
[{'price': 100.0}]

```

Fixed some edge cases around defaults for Array columns.

```

def get_default():
    # This used to fail:
    return [datetime.time(hour=8, minute=0)]

class MyTable(Table):
    times = Array(Time(), default=get_default)

```

Fixed some deprecation warnings, and improved CockroachDB array tests.

18.34 1.8.0

Added the Cast function, for performing type conversion.

Here's an example, where we convert a timestamp to time:

```

>>> from piccolo.columns import Time
>>> from piccolo.query.functions import Cast

>>> await Concert.select(Cast(Concert.starts, Time()))
[{'starts': datetime.time(19, 0)}]

```

A new section was also added to the docs describing functions in more detail.

18.35 1.7.0

Arrays of Date / Time / Timestamp / Timestamptz now work in SQLite.

For example:

```
class MyTable(Table):
    times = Array(Time())
    dates = Array(Date())
    timestamps = Array(Timestamp())
    timestamps_tz = Array(Timestamptz())
```

18.36 1.6.0

Added support for a bunch of Postgres functions, like Upper, Lower, Length, and Ltrim. They can be used in select queries:

```
from piccolo.query.functions.string import Upper
>>> await Band.select(Upper(Band.name, alias="name"))
[{"name": "PYTHONISTAS"}]
```

And also in where clauses:

```
>>> await Band.select().where(Upper(Band.manager.name) == 'GUIDO')
[{"name": "Pythonistas"}]
```

18.37 1.5.2

Added an Album table to the playground, along with some other improvements.

Fixed a bug with the output (load_json=True) clause, when used on joined tables.

18.38 1.5.1

Fixed a bug with the CLI when reversing migrations (thanks to @metakot for reporting this).

Updated the ASGI templates (thanks to @tarsil for adding Lilya).

18.39 1.5.0

Lots of internal improvements, mostly to support new functionality in Piccolo Admin.

18.40 1.4.2

Improved how `ModelBuilder` handles recursive foreign keys.

18.41 1.4.1

Fixed an edge case with auto migrations.

If starting from a table like this, with a custom primary key column:

```
class MyTable(Table):
    id = UUID(primary_key=True)
```

When a foreign key is added to the table which references itself:

```
class MyTable(Table):
    id = UUID(primary_key=True)
    fk = ForeignKey("self")
```

The auto migrations could fail in some situations.

18.42 1.4.0

Improved how `create_pydantic_model` handles `Array` columns:

- Multidimensional arrays (e.g. `Array(Array(Integer))`) have more accurate types.
- `Array(Email())` now validates that each item in the list is an email address.
- `Array(VarChar(length=10))` now validates that each item is the correct length (i.e. 10 in this example).

18.42.1 Other changes

Some Pylance errors were fixed in the codebase.

18.43 1.3.2

Fixed a bug with nested array columns containing `BigInt`. For example:

```
class MyTable(Table):
    my_column = Array(Array(BigInt))
```

Thanks to [@AmazingAkai](#) for reporting this issue.

18.44 1.3.1

Fixed a bug with foreign keys which reference `BigSerial` primary keys. Thanks to [@Abdelhadi92](#) for reporting this issue.

18.45 1.3.0

Added the piccolo user `list` command - a quick and convenient way of listing Piccolo Admin users from the command line.

`ModelBuilder` now creates timezone aware `datetime` objects for `Timestamptz` columns.

Updated the ASGI templates.

SQLite auto migrations are now allowed. We used to raise an exception, but now we output a warning instead. While SQLite auto migrations aren't as feature rich as Postgres, they work fine for simple use cases.

18.46 1.2.0

There's now an alternative syntax for joins, which works really well with static type checkers like Mypy and Pylance.

The traditional syntax (which continues to work as before):

```
# Get the band name, and the manager's name from a related table  
await Band.select(Band.name, Band.manager.name)
```

The alternative syntax is as follows:

```
await Band.select(Band.name, Band.manager.!.name)
```

Note how we use `!.` instead of `.` after a `ForeignKey`.

This offers a considerably better static typing experience. In the above example, type checkers know that `Band.manager.!.name` refers to the `name` column on the `Manager` table. This means typos can be detected, and code navigation is easier.

18.46.1 Other changes

- Improve static typing for `get_related`.
 - Added support for the `esmerald` ASGI framework.
-

18.47 1.1.1

Piccolo allows the user to specify savepoint names which are used in transactions. For example:

```
async with DB.transaction() as transaction:  
    await Band.insert(Band(name='Pythonistas'))  
  
    # Passing in a savepoint name is optional:  
    savepoint_1 = await transaction.savepoint('savepoint_1')  
  
    await Band.insert(Band(name='Terrible band'))  
  
    # Oops, I made a mistake!  
    await savepoint_1.rollback_to()
```

Postgres doesn't allow us to parameterise savepoint names, which means there's a small chance of SQL injection, if for some reason the savepoint names were generated from end-user input. Even though the likelihood is very low, it's best to be safe. We now validate the savepoint name, to make sure it can only contain certain safe characters. Thanks to @Skelmis for making this change.

18.48 1.1.0

Added support for Python 3.12.

Modified `create_pydantic_model`, so additional information is returned in the JSON schema to distinguish between `Timestamp` and `Timestamptz` columns. This will be used for future Piccolo Admin enhancements.

18.49 1.0.0

Piccolo v1 is now available!

We migrated to Pydantic v2, and also migrated Piccolo Admin to Vue 3, which puts the project in a good place moving forward.

We don't anticipate any major issues for people who are upgrading. If you encounter any bugs let us know.

Make sure you have v1 of Piccolo, Piccolo API, and Piccolo Admin.

18.50 1.0a3

Namespaced all custom values we added to Pydantic's JSON schema for easier maintenance.

18.51 1.0a2

All of the changes from 0.120.0 merged into the v1 branch.

18.52 0.121.0

Modified the `BaseUser.login` logic so all code paths take the same time. Thanks to @Skelmis for this.

18.53 0.120.0

Improved how `ModelBuilder` generates JSON data.

The number of password hash iterations used in `BaseUser` has been increased to keep pace with the latest guidance from OWASP - thanks to @Skelmis for this.

Fixed a bug with auto migrations when the table is in a schema.

18.54 1.0a1

Initial alpha release of Piccolo v1, with Pydantic v2 support.

18.55 0.119.0

ModelBuilder now works with LazyTableReference (which is used when we have circular references caused by a ForeignKey).

With this table:

```
class Band(Table):
    manager = ForeignKey(
        LazyTableReference(
            'Manager',
            module_path='some.other.folder.tables'
        )
    )
```

We can now create a dynamic test fixture:

```
my_model = await ModelBuilder.build(Band)
```

18.56 0.118.0

If you have lots of Piccolo apps, you can now create auto migrations for them all in one go:

```
piccolo migrations new all --auto
```

Thanks to @hoosnick for suggesting this new feature.

The documentation for running migrations has also been improved, as well as improvements to the sorting of migrations based on their dependencies.

Support for Python 3.7 was dropped in this release as it's now end of life.

18.57 0.117.0

Version pinning Pydantic to v1, as v2 has breaking changes.

We will add support for Pydantic v2 in a future release.

Thanks to @sinisaos for helping with this.

18.58 0.116.0

18.58.1 Fixture formatting

When creating a fixture:

```
piccolo fixtures dump
```

The JSON output is now nicely formatted, which is useful because we can pipe it straight to a file, and commit it to Git without having to manually run a formatter on it.

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

Thanks to @sinisaos for this.

18.58.2 Protected table names

We used to raise a `ValueError` if a table was called `user`.

```
class User(Table): # ValueError!
    ...
```

It's because `user` is already used by Postgres (e.g. try `SELECT user` or `SELECT * FROM user`).

We now emit a warning instead for these reasons:

- Piccolo wraps table names in quotes to avoid clashes with reserved keywords.
- Sometimes you're stuck with a table name from a pre-existing schema, and can't easily rename it.

18.58.3 Re-export WhereRaw

If you want to write raw SQL in your where queries you use `WhereRaw`:

```
>>> Band.select().where(WhereRaw('TRIM(name) = {}'.format('Pythonistas')))
```

You can now import it from `piccolo.query` to be consistent with `SelectRaw` and `OrderByRaw`.

```
from piccolo.query import WhereRaw
```

18.59 0.115.0

18.59.1 Fixture upserting

Fixtures can now be upserted. For example:

```
piccolo fixtures load my_fixture.json --on_conflict='DO UPDATE'
```

The options are:

- `DO NOTHING`, meaning any rows with a matching primary key will be left alone.
- `DO UPDATE`, meaning any rows with a matching primary key will be updated.

This is really useful, as you can now edit fixtures and load them multiple times without getting foreign key constraint errors.

18.59.2 Schema fixes

We recently added support for schemas, for example:

```
class Band(Table, schema='music'):
    ...
```

This release contains:

- A fix for migrations when changing a table's schema back to 'public' (thanks to @sinisaos for discovering this).
- A fix for M2M queries, when the tables are in a schema other than 'public' (thanks to @quinnalfaro for reporting this).

18.59.3 Added distinct method to count queries

We recently added support for COUNT DISTINCT queries. The syntax is:

```
await Concert.count(distinct=[Concert.start_date])
```

The following alternative syntax now also works (just to be consistent with other queries like select):

```
await Concert.count().distinct([Concert.start_date])
```

18.60 0.114.0

count queries can now return the number of distinct rows. For example, if we have this table:

```
class Concert(Table):
    band = Varchar()
    start_date = Date()
```

With this data:

band	start_date
Pythonistas	2023-01-01
Pythonistas	2023-02-03
Rustaceans	2023-01-01

We can easily get the number of unique concert dates:

```
>>> await Concert.count(distinct=[Concert.start_date])
2
```

We could have just done this instead:

```
len(await Concert.select(Concert.start_date).distinct())
```

But it's far less efficient when you have lots of rows, because all of the distinct rows need to be returned from the database.

Also, the docs for the count query, aggregate functions, and group_by clause were significantly improved.

Many thanks to @lqmanh and @sinisaos for their help with this.

18.61 0.113.0

If Piccolo detects a renamed table in an auto migration, it asks the user for confirmation. When lots of tables have been renamed, Piccolo is now more intelligent about when to ask for confirmation. Thanks to @sumitsharansatsangi for suggesting this change, and @sinisaos for reviewing.

Also, fixed the type annotations for `MigrationManager.add_table`.

18.62 0.112.1

Fixed a bug with serialising table classes in migrations.

18.63 0.112.0

Added support for schemas in Postgres and CockroachDB.

For example:

```
class Band(Table, schema="music"):
    ...
```

When creating the table, the schema will be created automatically if it doesn't already exist.

```
await Band.create_table()
```

It also works with migrations. If we change the schema value for the table, Piccolo will detect this, and create a migration for moving it to the new schema.

```
class Band(Table, schema="music_2"):
    ...

# Piccolo will detect that the table needs to be moved to a new schema.
>>> piccolo migrations new my_app --auto
```

18.64 0.111.1

Fixing a bug with `ModelBuilder` and `Decimal / Numeric` columns.

18.65 0.111.0

Added the `on_conflict` clause for `insert` queries. This enables **upserts**.

For example, here we insert some bands, and if they already exist then do nothing:

```
await Band.insert(
    Band(name='Pythonistas'),
    Band(name='Rustaceans'),
    Band(name='C-Sharps'),
).on_conflict(action='DO NOTHING')
```

Here we insert some albums, and if they already exist then we update the price:

```
await Album.insert(
    Album(title='OK Computer', price=10.49),
    Album(title='Kid A', price=9.99),
    Album(title='The Bends', price=9.49),
).on_conflict(
    action='DO UPDATE',
    target=Album.title,
    values=[Album.price]
)
```

Thanks to @sinisaos for helping with this.

18.66 0.110.0

18.66.1 ASGI frameworks

The ASGI frameworks in `piccolo asgi new` have been updated. `starlite` has been renamed to `litestar`. Thanks to @sinisaos for this.

18.66.2 ModelBuilder

Generic types are now used in `ModelBuilder`.

```
# mypy knows this is a `Band` instance:
band = await ModelBuilder.build(Band)
```

18.66.3 DISTINCT ON

Added support for `DISTINCT ON` queries. For example, here we fetch the most recent album for each band:

```
>>> await Album.select().distinct(
...     on=[Album.band]
... ).order_by(
...     Album.band
... ).order_by(
...     Album.release_date,
...     ascending=False
... )
```

Thanks to @sinisaos and @williamflaherty for their help with this.

18.67 0.109.0

Joins are now possible without foreign keys using `join_on`.

For example:

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

class Band(Table):
    name = Varchar()
    manager_name = Varchar()

>>> await Band.select(
...     Band.name,
...     Band.manager_name.join_on(Manager.name).email
... )
```

18.68 0.108.0

Added support for savepoints within transactions.

```
async with DB.transaction() as transaction:
    await Manager.objects().create(name="Great manager")
    savepoint = await transaction.savepoint()
    await Manager.objects().create(name="Great manager")
    await savepoint.rollback_to()
# Only the first manager will be inserted.
```

The behaviour of nested context managers has also been changed slightly.

```
async with DB.transaction():
    async with DB.transaction():
        # This used to raise an exception
```

We no longer raise an exception if there are nested transaction context managers, instead the inner ones do nothing.

If you want the existing behaviour:

```
async with DB.transaction():
    async with DB.transactiona(allow_nested=False):
        # TransactionError!
```

18.69 0.107.0

Added the `log_responses` option to the database engines. This makes the engine print out the raw response from the database for each query, which is useful during debugging.

```
# piccolo_conf.py

DB = PostgresEngine(
    config={'database': 'my_database'},
    log_queries=True,
    log_responses=True
)
```

We also updated the Starlite ASGI template - it now uses the new import paths (thanks to @sinisaos for this).

18.70 0.106.0

Joins now work within update queries. For example:

```
await Band.update({
    Band.name: 'Amazing Band'
}).where(
    Band.manager.name == 'Guido'
)
```

Other changes:

- Improved the template used by piccolo app new when creating a new Piccolo app (it now uses table_finder).
-

18.71 0.105.0

Improved the performance of select queries with complex joins. Many thanks to @powellnorma and @sinisaos for their help with this.

18.72 0.104.0

Major improvements to Piccolo's typing / auto completion support.

For example:

```
>>> bands = await Band.objects() # List[Band]

>>> band = await Band.objects().first() # Optional[Band]

>>> bands = await Band.select().output(as_json=True) # str
```

18.73 0.103.0

18.73.1 SelectRaw

This allows you to access features in the database which aren't exposed directly by Piccolo. For example, Postgres functions:

```

from piccolo.query import SelectRaw

>>> await Band.select(
...     Band.name,
...     SelectRaw("log(popularity) AS log_popularity")
... )
[{'name': 'Pythonistas', 'log_popularity': 3.0}]

```

18.73.2 Large fixtures

Piccolo can now load large fixtures using `piccolo fixtures load`. The rows are inserted in batches, so the database adapter doesn't raise any errors.

18.74 0.102.0

18.74.1 Migration file names

The naming convention for migrations has changed slightly. It used to be just a timestamp - for example:

```
2021-09-06T13-58-23-024723.py
```

By convention Python files should start with a letter, and only contain a-z, 0-9 and `_`, so the new format is:

```
my_app_2021_09_06T13_58_23_024723.py
```

Note

You can name a migration file anything you want (it's the ID value inside it which is important), so this change doesn't break anything.

18.74.2 Enhanced Pydantic configuration

We now expose all of Pydantic's configuration options to `create_pydantic_model`:

```

class MyPydanticConfig(pydantic.BaseConfig):
    extra = 'forbid'

model = create_pydantic_model(
    table=MyTable,
    pydantic_config_class=MyPydanticConfig
)

```

Thanks to @waldner for this.

18.74.3 Other changes

- Fixed a bug with `get_or_create` and null columns (thanks to @powellnorma for reporting this issue).
 - Updated the Starlite ASGI template, so it uses the latest syntax for mounting Piccolo Admin (thanks to @sinisaos for this, and the Starlite team).
-

18.75 0.101.0

`piccolo fixtures load` is now more intelligent about how it loads data, to avoid foreign key constraint errors.

18.76 0.100.0

Array columns now support choices.

```
class Ticket(Table):
    class Extras(str, enum.Enum):
        drink = "drink"
        snack = "snack"
        program = "program"

    extras = Array(Varchar(), choices=Extras)
```

We can then use the Enum in our queries:

```
>>> await Ticket.insert(
...     Ticket(extras=[Extras.drink, Extras.snack]),
...     Ticket(extras=[Extras.program]),
... )
```

This will also be supported in Piccolo Admin in the next release.

18.77 0.99.0

You can now use the `returning` clause with `delete` queries.

For example:

```
>>> await Band.delete().where(Band.popularity < 100).returning(Band.name)
[{'name': 'Terrible Band'}, {'name': 'Awful Band'}]
```

This also means you can count the number of deleted rows:

```
>>> len(await Band.delete().where(Band.popularity < 100).returning(Band.id))
2
```

Thanks to @waldner for adding this feature.

18.78 0.98.0

18.78.1 SQLite TransactionType

You can now specify the transaction type for SQLite.

This is useful when using SQLite in production, as it's possible to get database locked errors if you're running lots of transactions concurrently, and don't use the correct transaction type.

In this example we use an IMMEDIATE transaction:

```

from piccolo.engine.sqlite import TransactionType

async with Band._meta.db.transaction(
    transaction_type=TransactionType.immediate
):
    band = await Band.objects().get_or_create(Band.name == 'Pythonistas')
    ...

```

We've added a [new tutorial](#) which explains this in more detail, as well as other tips for using asyncio and SQLite together effectively.

Thanks to @powellnorma and @sinisaos for their help with this.

18.78.2 Other changes

- Fixed a bug with camelCase column names (we recommend using snake_case, but sometimes it's unavoidable when using Piccolo with an existing schema). Thanks to @sinisaos for this.
- Fixed a typo in the docs with raw queries - thanks to @StitiFatah for this.

18.79 0.97.0

Some big improvements to order_by clauses.

It's now possible to combine ascending and descending:

```

await Band.select(
    Band.name,
    Band.popularity
).order_by(
    Band.name
).order_by(
    Band.popularity,
    ascending=False
)

```

You can also order by anything you want using OrderByRaw:

```

from piccolo.query import OrderByRaw

await Band.select(
    Band.name
).order_by(

```

(continues on next page)

```
OrderByRaw('random()')
)
```

18.80 0.96.0

Added the `auto_update` argument to `Column`. Its main use case is columns like `modified_on` where we want the value to be updated automatically each time the row is saved.

```
class Band(Table):
    name = Varchar()
    popularity = Integer()
    modified_on = Timestamp(
        null=True,
        default=None,
        auto_update=datetime.datetime.now
    )

# The `modified_on` column will automatically be updated to the current
# timestamp:
>>> await Band.update({
...     Band.popularity: Band.popularity + 100
... }).where(
...     Band.name == 'Pythonistas'
... )
```

It works with `MyTable.update` and also when using the `save` method on an existing row.

18.81 0.95.0

Made improvements to the Piccolo playground.

- Syntax highlighting is now enabled.
- The example queries are now async (iPython supports top level await, so this works fine).
- You can optionally use your own iPython configuration `piccolo playground run --ipython_profile` (for example if you want a specific colour scheme, rather than the one we use by default).

Thanks to @haffi96 for this. See [PR 656](#).

18.82 0.94.0

Fixed a bug with `MyTable.objects().create()` and columns which are not nullable. Thanks to @metakot for reporting this issue.

We used to use `logging.getLogger(__file__)`, but as @Drapersniper pointed out, the Python docs recommend `logging.getLogger(__name__)`, so it has been changed.

18.83 0.93.0

- Fixed a bug with nullable JSON / JSONB columns and `create_pydantic_model` - thanks to @eneacosta for this fix.
 - Made the `Time` column type importable from `piccolo.columns`.
 - Python 3.11 is now supported.
 - Postgres 9.6 is no longer officially supported, as it's end of life, but Piccolo should continue to work with it just fine for now.
 - Improved docs for transactions, added docs for the `as_of` clause in CockroachDB (thanks to @gnat for this), and added docs for `add_raw_backwards`.
-

18.84 0.92.0

Added initial support for Cockroachdb (thanks to @gnat for this massive contribution).

Fixed Pylance warnings (thanks to @MiguelGuthridge for this).

18.85 0.91.0

Added support for Starlite. If you use `piccolo asgi new` you'll see it as an option for a router.

Thanks to @sinisaos for adding this, and @peterschutt for helping debug ASGI mounting.

18.86 0.90.0

Fixed an edge case, where a migration could fail if:

- 5 or more tables were being created at once.
- They all contained foreign keys to each other, as shown below.

```
class TableA(Table):
    pass

class TableB(Table):
    fk = ForeignKey(TableA)

class TableC(Table):
    fk = ForeignKey(TableB)

class TableD(Table):
    fk = ForeignKey(TableC)

class TableE(Table):
    fk = ForeignKey(TableD)
```

Thanks to @sumitsharansatsangi for reporting this issue.

18.87 0.89.0

Made it easier to access the Email columns on a table.

```
>>> MyTable._meta.email_columns
[MyTable.email_column_1, MyTable.email_column_2]
```

This was added for Piccolo Admin.

18.88 0.88.0

Fixed a bug with migrations - when using `db_column_name` it wasn't being used in some alter statements. Thanks to @theelderbeever for reporting this issue.

```
class Concert(Table):
    # We use `db_column_name` when the column name is problematic - e.g. if
    # it clashes with a Python keyword.
    in_ = Varchar(db_column_name='in')
```

18.89 0.87.0

When using `get_or_create` with `prefetch` the behaviour was inconsistent - it worked as expected when the row already existed, but `prefetch` wasn't working if the row was being created. This now works as expected:

```
>>> band = Band.objects(Band.manager).get_or_create(
...     (Band.name == "New Band 2") & (Band.manager == 1)
... )

>>> band.manager
<Manager: 1>
>>> band.manager.name
"Mr Manager"
```

Thanks to @backwardspsy for reporting this issue.

18.90 0.86.0

Added the Email column type. It's basically identical to Varchar, except that when we use `create_pydantic_model` we add email validation to the generated Pydantic model.

```
from piccolo.columns.column_types import Email
from piccolo.table import Table
from piccolo.utils.pydantic import create_pydantic_model

class MyTable(Table):
    email = Email()
```

(continues on next page)

(continued from previous page)

```

model = create_pydantic_model(MyTable)

model(email="not a valid email")
# ValidationError!

```

Thanks to @sinisaos for implementing this feature.

18.91 0.85.1

Fixed a bug with migrations - when run backwards, raw was being called instead of raw_backwards. Thanks to @translunar for the fix.

18.92 0.85.0

You can now append items to an array in an update query:

```

await Ticket.update({
    Ticket.seat_numbers: Ticket.seat_numbers + [1000]
}).where(Ticket.id == 1)

```

Currently Postgres only. Thanks to @sumitsharansatsangi for suggesting this feature.

18.93 0.84.0

You can now preview the DDL statements which will be run by Piccolo migrations.

```

piccolo migrations forwards my_app --preview

```

Thanks to @AliSayyah for adding this feature.

18.94 0.83.0

We added support for Postgres read-slaves a few releases ago, but the batch clause didn't support it until now. Thanks to @guruvignesh01 for reporting this issue, and @sinisaos for help implementing it.

```

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

```

18.95 0.82.0

Traditionally, when instantiating a `Table`, you passed in column values using kwargs:

```
>>> await Manager(name='Guido').save()
```

You can now pass in a dictionary instead, which makes it easier for static typing analysis tools like Mypy to detect typos.

```
>>> await Manager({Manager.name: 'Guido'}).save()
```

See [PR 565](#) for more info.

18.96 0.81.0

Added the `returning` clause to `insert` and `update` queries.

This can be used to retrieve data from the inserted / modified rows.

Here's an example, where we update the unpopular bands, and retrieve their names, in a single query:

```
>>> await Band.update({
...     Band.popularity: Band.popularity + 5
... }).where(
...     Band.popularity < 10
... ).returning(
...     Band.name
... )
[{'name': 'Bad sound band'}, {'name': 'Tone deaf band'}]
```

See [PR 564](#) and [PR 563](#) for more info.

18.97 0.80.2

Fixed a bug with `Combination.__str__`, which meant that when printing out a query for debugging purposes it wasn't showing correctly (courtesy [@destos](#)).

18.98 0.80.1

Fixed a bug with Piccolo Admin and `_get_related_readable`, which is used to show a human friendly identifier for a row, rather than just the ID.

Thanks to [@ethagnawl](#) and [@sinisaos](#) for their help with this.

18.99 0.80.0

There was a bug when doing joins with a `JSONB` column with `as_alias`.

```

class User(Table, tablename="my_user"):
    name = Varchar(length=120)
    config = JSONB(default={})

class Subscriber(Table, tablename="subscriber"):
    name = Varchar(length=120)
    user = ForeignKey(references=User)

async def main():
    # This was failing:
    await Subscriber.select(
        Subscriber.name,
        Subscriber.user.config.as_alias("config")
    )

```

Thanks to @Anton-Karpenko for reporting this issue.

Even though this is a bug fix, the minor version number has been bumped because the fix resulted in some refactoring of Piccolo's internals, so is a fairly big change.

18.100 0.79.0

Added a custom `__repr__` method to Table's metaclass. It's needed to improve the appearance of our Sphinx docs. See [issue 549](#) for more details.

18.101 0.78.0

Added the `callback` clause to `select` and `objects` queries (courtesy @backwardspy). For example:

```
>>> await Band.select().callback(my_callback)
```

The callback can be a normal function or async function, which is called when the query is successful. The callback can be used to modify the query's output.

It allows for some interesting and powerful code. Here's a very simple example where we modify the query's output:

```

>>> def get_uppercase_names() -> Select:
...     def make_uppercase(response):
...         return [{'name': i['name'].upper()} for i in response]
...
...     return Band.select(Band.name).callback(make_uppercase)

>>> await get_uppercase_names().where(Band.manager.name == 'Guido')
[{'name': 'PYTHONISTAS'}]

```

Here's another example, where we perform validation on the query's output:

```
>>> def get_concerts() -> Select:
...     def check_length(response):
...         if len(response) == 0:
...             raise ValueError('No concerts!')
...         return response
...
...     return Concert.select().callback(check_length)

>>> await get_concerts().where(Concert.band_1.name == 'Terrible Band')
ValueError: No concerts!
```

At the moment, callbacks are just triggered when a query is successful, but in the future other callbacks will be added, to hook into more of Piccolo's internals.

18.102 0.77.0

Added the `refresh` method. If you have an object which has gotten stale, and want to refresh it, so it has the latest data from the database, you can now do this:

```
# If we have an instance:
band = await Band.objects().first()

# And it has gotten stale, we can refresh it:
await band.refresh()
```

Thanks to @trondhindenes for suggesting this feature.

18.103 0.76.1

Fixed a bug with `atomic` when run async with a connection pool.

For example:

```
atomic = Band._meta.db.atomic()
atomic.add(query_1, query_1)
# This was failing:
await atomic.run()
```

Thanks to @Anton-Karpenko for reporting this issue.

18.104 0.76.0

18.104.1 create_db_tables / drop_db_tables

Added `create_db_tables` and `create_db_tables_sync` to replace `create_tables`. The problem was `create_tables` was sync only, and was inconsistent with the rest of Piccolo's API, which is async first. `create_tables` will continue to work for now, but is deprecated, and will be removed in version 1.0.

Likewise, `drop_db_tables` and `drop_db_tables_sync` have replaced `drop_tables`.

When calling `create_tables` / `drop_tables` within other async libraries (such as `ward`) it was sometimes unreliable - the best solution was just to make async versions of these functions. Thanks to [@backwardspy](#) for reporting this issue.

18.104.2 BaseUser password validation

We centralised the password validation logic in `BaseUser` into a method called `_validate_password`. This is needed by Piccolo API, but also makes it easier for users to override this logic if subclassing `BaseUser`.

18.104.3 More `run_sync` refinements

`run_sync`, which is the main utility function which Piccolo uses to run async code, has been further simplified for Python > v3.10 compatibility.

18.105 0.75.0

Changed how `piccolo.utils.sync.run_sync` works, to prevent a warning on Python 3.10. Thanks to [@Drapersniper](#) for reporting this issue.

Lots of documentation improvements - particularly around testing, and Docker deployment.

18.106 0.74.4

`piccolo schema generate` now outputs a warning when it can't detect the `ON DELETE` and `ON UPDATE` for a `ForeignKey`, rather than raising an exception. Thanks to [@theelderbeever](#) for reporting this issue.

`run_sync` doesn't use the connection pool by default anymore. It was causing issues when an app contained sync and async code. Thanks to [@WintonLi](#) for reporting this issue.

Added a tutorial to the docs for using Piccolo with an existing project and database. Thanks to [@virajkanwade](#) for reporting this issue.

18.107 0.74.3

If you had a table containing an array of `BigInt`, then migrations could fail:

```
from piccolo.table import Table
from piccolo.columns.column_types import Array, BigInt

class MyTable(Table):
    my_column = Array(base_column=BigInt())
```

It's because the `BigInt` base column needs access to the parent table to know if it's targeting Postgres or SQLite. See [PR 501](#).

Thanks to [@cheesycod](#) for reporting this issue.

18.108 0.74.2

If a user created a custom Column subclass, then migrations would fail. For example:

```
class CustomColumn(VarChar):
    def __init__(self, custom_arg: str = '', *args, **kwargs):
        self.custom_arg = custom_arg
        super().__init__(*args, **kwargs)

    @property
    def column_type(self):
        return 'VARCHAR'
```

See PR 497. Thanks to @WintonLi for reporting this issue.

18.109 0.74.1

When using `pip install piccolo[all]` on Windows it would fail because uvloop isn't supported. Thanks to @jack1142 for reporting this issue.

18.110 0.74.0

We've had the ability to bulk modify rows for a while. Here we append '!!!' to each band's name:

```
>>> await Band.update({Band.name: Band.name + '!!!'}, force=True)
```

It only worked for some columns - Varchar, Text, Integer etc.

We now allow Date, Timestamp, Timestamptz and Interval columns to be bulk modified using a `timedelta`. Here we modify each concert's start date, so it's one day later:

```
>>> await Concert.update(
...     {Concert.starts: Concert.starts + timedelta(days=1)},
...     force=True
... )
```

Thanks to @theelderbeever for suggesting this feature.

18.111 0.73.0

You can now specify extra nodes for a database. For example, if you have a read replica.

```
DB = PostgresEngine(
    config={'database': 'main_db'},
    extra_nodes={
        'read_replica_1': PostgresEngine(
            config={
```

(continues on next page)

(continued from previous page)

```
        'database': 'main_db',
        'host': 'read_replica_1.my_db.com'
    }
)
)
```

And can then run queries on these other nodes:

```
>>> await MyTable.select().run(node="read_replica_1")
```

See [PR 481](#). Thanks to @dashsatisf for suggesting this feature.

Also, the `targ` library has been updated so it tells users about the `--trace` argument which can be used to get a full traceback when a CLI command fails.

18.112 0.72.0

Fixed typos with `drop_constraints`. Courtesy @smythp.

Lots of documentation improvements, such as fixing Sphinx's autodoc for the `Array` column.

`AppConfig` now accepts a `pathlib.Path` instance. For example:

```
# piccolo_app.py

import pathlib

APP_CONFIG = AppConfig(
    app_name="blog",
    migrations_folder_path=pathlib.Path(__file__) / "piccolo_migrations"
)
```

Thanks to @theelderbeever for recommending this feature.

18.113 0.71.1

Fixed a bug with `ModelBuilder` and nullable columns (see [PR 462](#)). Thanks to @fiolet069 for reporting this issue.

18.114 0.71.0

The `ModelBuilder` class, which is used to generate mock data in tests, now supports `Array` columns. Courtesy @backwardspy.

Lots of internal code optimisations and clean up. Courtesy @yezz123.

Added docs for troubleshooting common MyPy errors.

Also thanks to @adriangb for helping us with our dependency issues.

18.115 0.70.1

Fixed a bug with auto migrations. If renaming multiple columns at once, it could get confused. Thanks to @theelder-beever for reporting this issue, and @sinisaos for helping to replicate it. See [PR 457](#).

18.116 0.70.0

We ran a profiler on the Piccolo codebase and identified some optimisations. For example, we were calling `self.querystring` multiple times in a method, rather than assigning it to a local variable.

We also ran a linter which identified when list / set / dict comprehensions could be more efficient.

The performance is now slightly improved (especially when fetching large numbers of rows from the database).

Example query times on a MacBook, when fetching 1000 rows from a local Postgres database (using `await SomeTable.select()`):

- 8 ms without a connection pool
- 2 ms with a connection pool

As you can see, having a connection pool is the main thing you can do to improve performance.

Thanks to @AliSayyah for all his work on this.

18.117 0.69.5

Made improvements to `piccolo schema generate`, which automatically generates Piccolo `Table` classes from an existing database.

There were situations where it would fail ungracefully when it couldn't parse an index definition. It no longer crashes, and we print out the problematic index definitions. See [PR 449](#). Thanks to @gmos for originally reporting this issue.

We also improved the error messages if schema generation fails for some reason by letting the user know which table caused the error. Courtesy @AliSayyah.

18.118 0.69.4

We used to raise a `ValueError` if a column was both `null=False` and `default=None`. This has now been removed, as there are situations where it's valid for columns to be configured that way. Thanks to @gmos for suggesting this change.

18.119 0.69.3

The `where` clause now raises a `ValueError` if a boolean value is passed in by accident. This was possible in the following situation:

```
await Band.select().where(Band.has_drummer is None)
```

Piccolo can't override the `is` operator because Python doesn't allow it, so `Band.has_drummer is None` will always equal `False`. Thanks to @trondhindenes for reporting this issue.

We've also put a lot of effort into improving documentation throughout the project.

18.120 0.69.2

- Lots of documentation improvements, including how to customise `BaseUser` (courtesy @sinisaos).
 - Fixed a bug with creating indexes when the column name clashes with a SQL keyword (e.g. 'order'). See [PR 433](#). Thanks to @wmshort for reporting this issue.
 - Fixed an issue where some slots were incorrectly configured (courtesy @ariebovenberg). See [PR 426](#).
-

18.121 0.69.1

Fixed a bug with auto migrations which rename columns - see [PR 423](#). Thanks to @theelderbeever for reporting this, and @sinisaos for help investigating.

18.122 0.69.0

Added `Xpresso` as a supported ASGI framework when using `piccolo asgi new` to generate a web app.

Thanks to @sinisaos for adding this template, and @adriangb for reviewing.

We also took this opportunity to update our FastAPI and BlackSheep ASGI templates.

18.123 0.68.0

18.123.1 Update queries without a where clause

If you try and perform an update query without a `where` clause you will now get an error:

```
>>> await Band.update({Band.name: 'New Band'})
UpdateError
```

If you want to update all rows in the table, you can still do so, but you must pass `force=True`.

```
>>> await Band.update({Band.name: 'New Band'}, force=True)
```

This is similar to `delete` queries, which require a `where` clause or `force=True`.

It was pointed out by @theelderbeever that an accidental mass update is almost as bad as a mass deletion, which is why this safety measure has been added.

See [PR 412](#).

 **Warning**

This is a breaking change. If you're doing update queries without a where clause, you will need to add `force=True`.

18.123.2 JSONB improvements

Fixed some bugs with nullable JSONB columns. A value of `None` is now stored as `null` in the database, instead of the JSON string `'null'`. Thanks to @theelderbeever for reporting this.

See [PR 413](#).

18.124 0.67.0

18.124.1 create_user

`BaseUser` now has a `create_user` method, which adds some extra password validation vs just instantiating and saving `BaseUser` directly.

```
>>> await BaseUser.create_user(username='bob', password='abc123XYZ')
<BaseUser: 1>
```

We check that passwords are a reasonable length, and aren't already hashed. See [PR 402](#).

18.124.2 async first

All of the docs have been updated to show the async version of queries.

For example:

```
# Previous:
Band.select().run_sync()

# Now:
await Band.select()
```

Most people use Piccolo in async apps, and the playground supports top level `await`, so you can just paste in `await Band.select()` and it will still work. See [PR 407](#).

We decided to use `await Band.select()` instead of `await Band.select().run()`. Both work, and have their merits, but the simpler version is probably easier for newcomers.

18.125 0.66.1

In Piccolo you can print out any query to see the SQL which will be generated:

```
>>> print(Band.select())
SELECT "band"."id", "band"."name", "band"."manager", "band"."popularity" FROM band
```

It didn't represent `UUID` and `datetime` values correctly, which is now fixed (courtesy @theelderbeever). See [PR 405](#).

18.126 0.66.0

Using descriptors to improve MyPy support (PR 399).

MyPy is now able to correctly infer the type in lots of different scenarios:

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

# MyPy knows this is a Varchar
Band.name

band = Band()
band.name = "Pythonistas" # MyPy knows we can assign strings when it's a class instance
band.name # MyPy knows we will get a string back

band.name = 1 # MyPy knows this is an error, as we should only be allowed to assign
↳ strings
```

18.127 0.65.1

Fixed bug with BaseUser and Piccolo API.

18.128 0.65.0

The BaseUser table hashes passwords before storing them in the database.

When we create a fixture from the BaseUser table (using `piccolo fixtures dump`), it looks something like:

```
{
  "id": 11,
  "username": "bob",
  "password": "pbkdf2_sha256$10000$abc123",
}
```

When we load the fixture (using `piccolo fixtures load`) we need to be careful in case BaseUser tries to hash the password again (it would then be a hash of a hash, and hence incorrect). We now have additional checks in place to prevent this.

Thanks to @mrbazzan for implementing this, and @sinisaos for help reviewing.

18.129 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.

18.130 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).

18.131 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

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.

18.132 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.

18.133 0.62.2

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

18.134 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.

18.135 0.62.0

Added Many-To-Many support.

```

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

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.

18.136 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.

18.137 0.61.1

18.137.1 Nicer ASGI template

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

18.137.2 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.

18.137.3 Fix Pylance error

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

18.138 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.

18.139 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.

18.140 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.

18.141 0.60.0

18.141.1 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.

18.141.2 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))
```

18.142 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.
-

18.143 0.58.0

18.143.1 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.

18.143.2 Internal code refactoring

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

18.143.3 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")
```

18.143.4 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.

18.143.5 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.

18.143.6 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 @OscarB.

18.144 0.57.0

Added Python 3.10 support (courtesy @kennethcheo).

18.145 0.56.0

18.145.1 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.

18.145.2 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()
```

(continues on next page)

(continued from previous page)

```

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.

18.145.3 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.

18.146 0.55.0

18.146.1 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.

18.146.2 ColumnMeta edge cases

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

18.146.3 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.

18.146.4 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.

18.146.5 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.

18.147 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'}
```

18.148 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}, ...]
```

18.149 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 `BigInt` column type (courtesy @aliereno).

Added GitHub issue templates (courtesy @AbhijithGanesh).

18.150 0.51.1

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

18.151 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.

18.152 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).

18.153 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).

18.154 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.

18.155 0.47.0

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

18.156 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.

18.157 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.

18.158 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.

18.159 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'}}
```

18.160 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.

18.161 0.43.0

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

18.162 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()
```

18.163 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.

18.164 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()
```

18.165 0.40.1

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

18.166 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.

18.167 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 @aminalae and @sinisaos for investigating edge cases.

18.168 0.38.2

Removed problematic type hint which assumed `pytest` was installed.

18.169 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.

18.170 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.

18.171 0.37.0

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

18.172 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.

18.173 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 @aminalae). 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()
```

18.174 0.34.0

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

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

18.175 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](#)).
-

18.176 0.33.0

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

18.177 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).

18.178 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]`).

18.179 0.30.0

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

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.

18.180 0.29.0

The user can now specify the primary key column (courtesy [@aminalae](#)). 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](#)).

18.181 0.28.0

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

18.182 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.

18.183 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`.

18.184 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.
-

18.185 0.24.1

A bug fix for migrations. See [issue 123](#) for more information.

18.186 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}
```

18.187 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).

18.188 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 explicitly reference the `id`.
-

18.189 0.21.2

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

18.190 0.21.1

Fix missing imports in FastAPI and Starlette app templates.

18.191 0.21.0

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

18.192 0.20.0

Added choices option to Column.

18.193 0.19.1

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

18.194 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
```

18.195 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).
-

18.196 0.18.3

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

18.197 0.18.2

Added support for filtering `Array` columns.

18.198 0.18.1

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

18.199 0.18.0

- Refactored `forwards` and `backwards` commands for migrations, to make them easier to run programmatically.
 - 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.
-

18.200 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.

18.201 0.17.4

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

18.202 0.17.3

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

18.203 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.
-

18.204 0.17.1

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

18.205 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.
-

18.206 0.16.5

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

18.207 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`.
-

18.208 0.16.3

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

18.209 0.16.2

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

18.210 0.16.1

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

18.211 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.
-

18.212 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.

18.213 0.15.0

Added Bytea / Blob column type.

18.214 0.14.13

Fixing a bug with migrations which drop column defaults.

18.215 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.
-

18.216 0.14.11

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

18.217 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 constraints.
-

18.218 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.

18.219 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.
-

18.220 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.
-

18.221 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.
-

18.222 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`
-

18.223 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.
-

18.224 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.
-

18.225 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.
-

18.226 0.14.1

Fixing a bug with dropping tables in auto migrations.

18.227 0.14.0

Added `Interval` column type.

18.228 0.13.5

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

18.229 0.13.4

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

18.230 0.13.3

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

18.231 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.
-

18.232 0.13.1

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

18.233 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.

18.234 0.12.6

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

18.235 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.
-

18.236 0.12.4

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

18.237 0.12.3

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

18.238 0.12.2

Fixing bug when sorting `extra_definitions` in auto migrations.

18.239 0.12.1

- Fixed typos.
 - Bumped requirements.
-

18.240 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.
-

18.241 0.11.8

ASGI template fix.

18.242 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.
-

18.243 0.11.6

Added support for FastAPI to `piccolo asgi new`.

18.244 0.11.5

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

18.245 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.
-

18.246 0.11.3

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

18.247 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.
-

18.248 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.
-

18.249 0.11.0

Added `Numeric` and `Real` column types.

18.250 0.10.8

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

18.251 0.10.7

Improving release script.

18.252 0.10.6

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

18.253 0.10.5

Auto migrations can now run backwards.

18.254 0.10.4

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

18.255 0.10.3

Adding missing jinja templates to `setup.py`.

18.256 0.10.2

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

18.257 0.10.1

Fixing bug with enum default values.

18.258 0.10.0

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

18.259 0.9.3

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

18.260 0.9.2

`.first()` bug fix.

18.261 0.9.1

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

18.262 0.9.0

Added support for auto migrations.

18.263 0.8.3

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

18.264 0.8.2

Fixing release issue.

18.265 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.

18.266 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.

18.267 0.7.7

Column sequencing matches the definition order.

18.268 0.7.6

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

18.269 0.7.5

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

18.270 0.7.4

Bug fixes and dependency updates.

18.271 0.7.3

Adding missing `__init__.py` file.

18.272 0.7.2

Changed migration import paths.

18.273 0.7.1

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

18.274 0.7.0

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

18.275 0.6.1

Adding missing `__init__.py` files.

18.276 0.6.0

Moved BaseUser. Migration refactor.

18.277 0.5.2

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

18.278 0.5.1

Added batch support.

18.279 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.

18.280 0.4.1

SQLite is now returning datetime objects for timestamp fields.

18.281 0.4.0

Refactored to improve code completion, along with bug fixes.

18.282 0.3.7

Allowing Update queries in SQLite.

18.283 0.3.6

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

18.284 0.3.5

Renamed User to BaseUser.

18.285 0.3.4

Added `ilike`.

18.286 0.3.3

Added value types to columns.

18.287 0.3.2

Default values infer the engine type.

18.288 0.3.1

Update click version.

18.289 0.3

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

18.290 0.2

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

18.291 0.1.2

Allowing joins across multiple tables.

18.292 0.1.1

Added playground.

HELP

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

20.1 Table

```
class piccolo.table.Table(_data: dict[Column, Any] | None = None, _ignore_missing: bool = False,
                          _exists_in_db: bool = False, **kwargs)
```

The class represents a database table. An instance represents a row.

The constructor can be used to assign column values.

Note

The `_data`, `_ignore_missing`, and `_exists_in_db` arguments are prefixed with an underscore to help prevent a clash with a column name which might be passed in via kwargs.

Parameters

- **_data** – There’s two ways of passing in the data for each column. Firstly, you can use kwargs:

```
Band(name="Pythonistas")
```

Secondly, you can pass in a dictionary which maps column classes to values:

```
Band({Band.name: 'Pythonistas'})
```

The advantage of this second approach is it’s more strongly typed, and linters such as flake8 or MyPy will more easily detect typos.

- **_ignore_missing** – If False a `ValueError` will be raised if any column values haven’t been provided.
- **_exists_in_db** – Used internally to track whether this row exists in the database.

```
add_m2m(*rows: Table, m2m: M2M, extra_column_values: dict[Column | str, Any] = {}) → 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
... )
[{'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."
    }
)
```

classmethod all_columns(*exclude: Sequence[str | Column] | None = None*) → list[Column]

Used in conjunction with select queries. Just as we can use `all_columns` to retrieve all of the columns from a related table, we can also use it at the root of our query to get all of the columns for the root table. For example:

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

This is mostly useful when the table has a lot of columns, and typing them out by hand would be tedious.

Parameters

exclude – You can request all columns, except these.

classmethod all_related(*exclude: list[str | ForeignKey] | None = None*) → list[ForeignKey]

Used in conjunction with objects queries. Just as we can use `all_related` on a `ForeignKey`, you can also use it for the table at the root of the query, which will return each related row as a nested object. For example:

```
concert = await Concert.objects(
    Concert.all_related()
)

>>> concert.band_1
<Band: 1>
>>> concert.band_2
<Band: 2>
>>> concert.venue
<Venue: 1>
```

This is mostly useful when the table has a lot of foreign keys, and typing them out by hand would be tedious. It's equivalent to:

```
concert = await Concert.objects(
    Concert.venue,
    Concert.band_1,
    Concert.band_2
)
```

Parameters

exclude – You can request all columns, except these.

classmethod alter() → Alter

Used to modify existing tables and columns.

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

classmethod count(*column: Column | None = None, distinct: Sequence[Column] | None = None*) → Count

Count the number of matching rows:

```
await Band.count().where(Band.popularity > 1000)
```

Parameters

- **column** – If specified, just count rows where this column isn't null.
- **distinct** – Counts the number of distinct values for these columns. For example, if we have a concerts table:

```
class Concert(Table):
    band = Varchar()
    start_date = Date()
```

With this data:

band	start_date
Pythonistas	2023-01-01
Pythonistas	2023-02-03
Rustaceans	2023-01-01

Without the `distinct` argument, we get the count of all rows:

```
>>> await Concert.count()
3
```

To get the number of unique concert dates:

```
>>> await Concert.count(distinct=[Concert.start_date])
2
```

classmethod create_index(*columns: list[Column] | list[str], method: IndexMethod = IndexMethod.btree, if_not_exists: bool = False*) → CreateIndex

Create a table index. If multiple columns are specified, this refers to a multicolumn index, rather than multiple single column indexes.

```
await Band.create_index([Band.name])
```

classmethod create_table(*if_not_exists=False, only_default_columns=False, auto_create_schema: bool = True*) → Create

Create table, along with all columns.

```
await Band.create_table()
```

classmethod delete(*force=False*) → Delete

Delete rows from the table.

```
await Band.delete().where(Band.name == 'Pythonistas')
```

Parameters

force – Unless set to True, deletions aren't allowed without a `where` clause, to prevent accidental mass deletions.

classmethod drop_index(*columns: list[Column] | list[str], if_exists: bool = True*) → DropIndex

Drop a table index. If multiple columns are specified, this refers to a multicolumn index, rather than multiple single column indexes.

```
await Band.drop_index([Band.name])
```

classmethod exists() → Exists

Use it to check if a row exists, not if the table exists.

```
await Band.exists().where(Band.name == 'Pythonistas')
```

classmethod from_dict(*data: dict[str, Any]*) → TableInstance

Used when loading fixtures. It can be overridden by subclasses in case they have specific logic / validation which needs running when loading fixtures.

get_m2m(*m2m: 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>]
```

classmethod get_readable() → Readable

Creates a readable representation of the row.

get_related(*foreign_key: ForeignKey[ReferencedTable]*) → GetRelated[ReferencedTable]

get_related(*foreign_key: str*) → GetRelated[Table]

Used to fetch a Table instance, for the target of a foreign key.

```
band = await Band.objects().first()
manager = await band.get_related(Band.manager)
>>> print(manager.name)
'Guido'
```

It can follow foreign keys multiple levels deep. For example, `Concert.band_1.manager`.

classmethod indexes() → Indexes

Returns a list of the indexes for this tables.

```
await Band.indexes()
```

classmethod insert(*rows: TableInstance) → Insert[TableInstance]

Insert rows into the database.

```
await Band.insert(
    Band(name="Pythonistas", popularity=500, manager=1)
)
```

classmethod objects(*prefetch: ForeignKey | list[ForeignKey]) → Objects[TableInstance]

Returns a list of table instances (each representing a row), which you can modify and then call 'save' on, or can delete by calling 'remove'.

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

pythonistas.name = 'Pythonistas Reborn'

await pythonistas.save()

# Or to remove it from the database:
await pythonistas.remove()
```

Parameters

prefetch – Rather than returning the primary key value of this related table, a nested object will be returned for the row on the related table.

```
# Without nested
band = await Band.objects().first()
>>> band.manager
1

# With nested
band = await Band.objects(Band.manager).first()
>>> band.manager
<Band 1>
```

property querystring: *QueryString*

Used when inserting rows.

classmethod raw(sql: str, *args: Any) → Raw

Execute raw SQL queries on the underlying engine - use with caution!

```
await Band.raw('select * from band')
```

Or passing in parameters:

```
await Band.raw("SELECT * FROM band WHERE name = {}", 'Pythonistas')
```

classmethod `ref(column_name: str) → Column`

Used to get a copy of a column from a table referenced by a `ForeignKey` column. It's unlikely an end user of this library will ever need to do this, but other libraries built on top of Piccolo may need this functionality.

```
Band.ref('manager.name')
```

refresh(*columns: Sequence[Column] | None = None, load_json: bool = False*) → *Refresh*

Used to fetch the latest data for this instance from the database. Modifies the instance in place, but also returns it as a convenience.

Parameters

- **columns** – If you only want to refresh certain columns, specify them here. Otherwise all columns are refreshed.
- **load_json** – Whether to load JSON / JSONB columns as objects, instead of just a string.

Example usage:

```
# Get an instance from the database.
instance = await Band.objects.first()

# Later on we can refresh this instance with the latest data
# from the database, in case it has gotten stale.
await instance.refresh()

# Alternatively, running it synchronously:
instance.refresh().run_sync()
```

remove() → Delete

A proxy to a delete query.

remove_m2m(*rows: Table, m2m: 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
... )
```

save(*columns: Sequence[str | Column] | None = None*) → Insert | Update

A proxy to an insert or update query.

Parameters

columns – Only the specified columns will be synced back to the database when doing an update. For example:

```
band = await Band.objects().first()
band.popularity = 2000
await band.save(columns=[Band.popularity])
```

If `columns=None` (the default) then all columns will be synced back to the database.

classmethod `select(*columns: Selectable | str, exclude_secrets=False) → Select`

Get data in the form of a list of dictionaries, with each dictionary representing a row.

These are all equivalent:

```
await Band.select().columns(Band.name)
await Band.select(Band.name)
await Band.select('name')
```

Parameters

exclude_secrets – If True, any columns with `secret=True` are omitted from the response. For example, we use this for the password column of `BaseUser`. Even though the passwords are hashed, you still don't want them being passed over the network if avoidable.

classmethod `table_exists() → TableExists`

Check if the table exists in the database.

```
await Band.table_exists()
```

to_dict(*columns: Column) → dict[str, Any]

A convenience method which returns a dictionary, mapping column names to values for this table instance.

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

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

If the columns argument is provided, only those columns are included in the output. It also works with column aliases.

```
>>> instance.to_dict(Manager.id, Manager.name.as_alias('title'))
{'id': 1, 'title': 'Guido'}
```

classmethod `update(values: dict[Column | str, Any] | None = None, force: bool = False, use_auto_update: bool = True, **kwargs) → Update`

Update rows.

All of the following work, though the first is preferable:

```
await Band.update(
    {Band.name: "Spamalot"}
).where(
    Band.name == "Pythonistas"
)

await Band.update(
    {"name": "Spamalot"}
).where(
    Band.name == "Pythonistas"
)

await Band.update(
```

(continues on next page)

(continued from previous page)

```

    name="Spamalot"
).where(
    Band.name == "Pythonistas"
)

```

Parameters

- **force** – Unless set to `True`, updates aren't allowed without a `where` clause, to prevent accidental mass overriding of data.
- **use_auto_update** – Whether to use the `auto_update` values on any columns. See the `auto_update` argument on [Column](#) for more information.

update_self(*values: dict[Column | str, Any]*) → UpdateSelf

This allows the user to update a single object - useful when the values are derived from the database in some way.

For example, if we have the following table:

```

class Band(Table):
    name = Varchar()
    popularity = Integer()

```

And we fetch an object:

```
>>> band = await Band.objects().get(name="Pythonistas")
```

We could use the typical syntax for updating the object:

```
>>> band.popularity += 1
>>> await band.save()
```

The problem with this, is what if another object has already incremented popularity? It would override the value.

Instead we can do this:

```
>>> await band.update_self({
...     Band.popularity: Band.popularity + 1
... })
```

This updates popularity in the database, and also sets the new value for popularity on the object.

20.2 SchemaManager

class piccolo.schema.SchemaManager(*db: Engine | None = None*)

A useful utility class for interacting with schemas.

Parameters

- **db** – Used to execute the database queries. If not specified, we try and import it from `piccolo_conf.py`.

create_schema(*schema_name: str*, * (Keyword-only parameters separator (PEP 3102)), *if_not_exists: bool = True*) → CreateSchema

Creates the specified schema:

```
>>> await SchemaManager().create_schema(schema_name="music")
```

Parameters

- **schema_name** – The name of the schema to create.
- **if_not_exists** – No error will be raised if the schema already exists.

drop_schema(*schema_name: str*, *, *if_exists: bool = True*, *cascade: bool = False*) → DropSchema

Drops the specified schema:

```
>>> await SchemaManager().drop_schema(schema_name="music")
```

Parameters

- **schema_name** – The name of the schema to drop.
- **if_exists** – No error will be raised if the schema doesn't exist.
- **cascade** – If True then it will automatically drop the tables within the schema.

list_schemas() → ListSchemas

Returns the name of each schema in the database:

```
>>> await SchemaManager().list_schemas()
['public', 'schema_1']
```

list_tables(*schema_name: str*) → ListTables

Returns the name of each table in the given schema:

```
>>> await SchemaManager().list_tables(schema_name="music")
['band', 'manager']
```

Parameters

schema_name – List the tables in this schema.

move_table(*table_name: str*, *new_schema: str*, *current_schema: str | None = None*) → MoveTable

Moves a table to a different schema:

```
>>> await SchemaManager().move_schema(
...     table_name='my_table',
...     new_schema='schema_1'
... )
```

Parameters

- **table_name** – The name of the table to move.
- **new_schema** – The name of the schema you want to move the table too.

Current_schema

If not specified, 'public' is assumed.

`rename_schema(schema_name: str, new_schema_name: str) → RenameSchema`

Rename the schema:

```
>>> await SchemaManager().rename_schema(  
...     schema_name="music",  
...     new_schema_name="music_info"  
... )
```

Parameters

- **schema_name** – The current name of the schema.
 - **new_schema_name** – What to rename the schema to.
-

20.3 Column

```
class piccolo.columns.base.Column(null: bool = False, primary_key: bool = False, unique: bool = False,  
                                  index: bool = False, index_method: IndexMethod = IndexMethod.btree,  
                                  required: bool = False, help_text: str | None = None, choices:  
                                  type[Enum] | None = None, db_column_name: str | None = None,  
                                  secret: bool = False, auto_update: Any = Ellipsis, **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 constraint 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")

>>> await MyTable.select(MyTable.class_)
[{'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)

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

- **auto_update** – Allows you to specify a value to set this column to each time it is updated (via `MyTable.update`, or `MyTable.save` on an existing row). A common use case is having a `modified_on` column.

```
class Band(Table):
    name = Varchar()
    popularity = Integer()
    # The value can be a function or static value:
    modified_on = Timestamp(auto_update=datetime.datetime.now)

# This will automatically set the `modified_on` column to the
# current timestamp, without having to explicitly set it:
>>> await Band.update({
...     Band.popularity: Band.popularity + 100
... }).where(Band.name == 'Pythonistas')
```

Note - this feature is implemented purely within the ORM. If you want similar functionality on the database level (i.e. if you plan on using raw SQL to perform updates), then you may be better off creating SQL triggers instead.

as_alias(*name: str*) → *Column*

Allows column names to be changed in the result of a select.

For example:

```
>>> await Band.select(Band.name.as_alias('title')).run()
{'title': 'Pythonistas'}
```

property ddl: *str*

Used when creating tables.

get_default_value() → *Any*

If the column has a default attribute, return it. If it's callable, return the response instead.

get_select_string(*engine_type: str*, *with_alias: bool = True*) → *QueryString*

How to refer to this column in a SQL query, taking account of any joins and aliases.

`get_sql_value(value: Any, delimiter: str = "") → str`

When using DDL statements, we can't parameterise the values. An example is when setting the default for a column. So we have to convert from the Python type to a string representation which we can include in our DDL statements.

Parameters

- **value** – The Python value to convert to a string usable in a DDL statement e.g. 1.
- **delimiter** – The string returned by this function is wrapped in delimiters, ready to be added to a DDL statement. For example: 'hello world'.

Returns

The string usable in the DDL statement e.g. '1'.

`ilike(value: str) → Where`

Only Postgres supports ILIKE. It's used for case insensitive matching.

For SQLite, it's just proxied to a LIKE query instead.

`is_not_null() → Where`

Can be used instead of `MyTable.column != None`, because some linters don't like a comparison to `None`.

`is_null() → Where`

Can be used instead of `MyTable.column == None`, because some linters don't like a comparison to `None`.

`join_on(column: Column) → ForeignKey`

Joins are typically performed via foreign key columns. For example, here we get the band's name and the manager's name:

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

class Band(Table):
    name = Varchar()
    manager = ForeignKey(Manager)

>>> await Band.select(Band.name, Band.manager.name)
```

The `join_on` method lets you join tables even when foreign keys don't exist, by joining on a column in another table.

For example, here we want to get the manager's email, but no foreign key exists:

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

class Band(Table):
    name = Varchar()
    manager_name = Varchar()

>>> await Band.select(
...     Band.name,
...     Band.manager_name.join_on(Manager.name).email
... )
```

like(*value: str*) → Where

Both SQLite and Postgres support LIKE, but they mean different things.

In Postgres, LIKE is case sensitive (i.e. 'foo' equals 'foo', but 'foo' doesn't equal 'Foo').

In SQLite, LIKE is case insensitive for ASCII characters (i.e. 'foo' equals 'Foo'). But not for non-ASCII characters. To learn more, see the docs:

https://sqlite.org/lang_expr.html#the_like_glob_regexp_and_match_operators

value_type

alias of `int`

class piccolo.columns.base.**ColumnKwargs**

Additional arguments which can be passed to `Column` from subclasses.

auto_update: `Any`

choices: `type[Enum] | None`

db_column_name: `str | None`

help_text: `str | None`

index: `bool`

index_method: `IndexMethod`

null: `bool`

primary_key: `bool`

required: `bool`

secret: `bool`

unique: `bool`

20.4 Aggregate functions

20.4.1 Count

class piccolo.query.methods.select.**Count**(*column: Column | None = None, distinct: Sequence[Column] | None = None, alias: str = 'count'*)

Used in Select queries, usually in conjunction with the `group_by` clause:

```
>>> await Band.select(
...     Band.manager.name.as_alias('manager_name'),
...     Count(alias='band_count')
... ).group_by(Band.manager)
[{'manager_name': 'Guido', 'count': 1}, ...]
```

It can also be used without the `group_by` clause (though you may prefer to the `Table.count` method instead, as it's more convenient):

```
>>> await Band.select(Count())
[{'count': 3}]
```

Parameters

- **column** – If specified, the count is for non-null values in that column.
- **distinct** – If specified, the count is for distinct values in those columns.
- **alias** – The name of the value in the response:

```
# These two are equivalent:

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

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

20.5 Refresh

class piccolo.query.methods.refresh.**Refresh**(*instance: Table, columns: Sequence[Column] | None = None, load_json: bool = False*)

Used to refresh *Table* instances with the latest data data from the database. Accessible via *refresh*.

Parameters

- **instance** – The instance to refresh.
- **columns** – Which columns to refresh - if not specified, then all columns are refreshed.
- **load_json** – Whether to load JSON / JSONB columns as objects, instead of just a string.

async run(*in_pool: bool = True, node: str | None = None*) → *Table*

Run it asynchronously. For example:

```
await my_instance.refresh().run()

# or for convenience:
await my_instance.refresh()
```

Modifies the instance in place, but also returns it as a convenience.

run_sync(*args, **kwargs) → *Table*

Run it synchronously. For example:

```
my_instance.refresh().run_sync()
```

20.6 LazyTableReference

```
class piccolo.columns.LazyTableReference(table_class_name: str, app_name: str | None = None,
                                        module_path: str | None = None)
```

Holds a reference to a *Table* subclass. Used to avoid circular dependencies in the `references` argument of *ForeignKey* columns.

Parameters

- **table_class_name** – The name of the Table subclass. For example, 'Manager'.
- **app_name** – If specified, the Table subclass is imported from a Piccolo app with the given name.
- **module_path** – If specified, the Table subclass is imported from this path. For example, 'my_app.tables'.

Hint

If the table is in the same file, you can pass in `__name__`.

20.7 Enums

20.7.1 Foreign Keys

```
class piccolo.columns.OnDelete(value)
```

Used by *ForeignKey* to specify the behaviour when a related row is deleted.

```
cascade = 'CASCADE'
```

```
no_action = 'NO ACTION'
```

```
restrict = 'RESTRICT'
```

```
set_default = 'SET DEFAULT'
```

```
set_null = 'SET NULL'
```

```
class piccolo.columns.OnUpdate(value)
```

Used by *ForeignKey* to specify the behaviour when a related row is updated.

```
cascade = 'CASCADE'
```

```
no_action = 'NO ACTION'
```

```
restrict = 'RESTRICT'
```

```
set_default = 'SET DEFAULT'
```

```
set_null = 'SET NULL'
```

20.7.2 Indexes

class piccolo.columns.indexes.**IndexMethod**(*value*)

Used to specify the index method for a *Column*.

btree = 'btree'

gin = 'gin'

gist = 'gist'

hash = 'hash'

20.8 Column defaults

20.8.1 Date

class piccolo.columns.defaults.**DateOffset**(*days: int*)

This makes the default value for a *Date* column the current date, but offset by a number of days.

For example, if you wanted the default to be tomorrow, you can specify `DateOffset(days=1)`:

```
class DiscountCode(Table):  
    expires = Date(default=DateOffset(days=1))
```

Parameters

days – The number of days to offset.

20.8.2 UUID

class piccolo.columns.defaults.**UUID4**

This makes the default value for a *UUID* column a randomly generated UUID v4 value. The advantage over using `uuid.uuid4()` from the standard library, is the default is set on the column definition in the database too.

class piccolo.columns.defaults.**UUID7**

This makes the default value for a *UUID* column a randomly generated UUID v7 value. Postgres ≥ 18 and Python ≥ 3.14 only. UUID7 can be more efficiently indexed than UUID, and inserts can be faster.

For this to work in older versions of Postgres, register an extension such as `pg_uuidv7`.

20.8.3 Timestamp

class piccolo.columns.defaults.**TimestampCustom**(*year: int = 2000, month: int = 1, day: int = 1, hour: int = 0, minute: int = 0, second: int = 0, microsecond: int = 0*)

class piccolo.columns.defaults.**TimestampOffset**(*days: int = 0, hours: int = 0, minutes: int = 0, seconds: int = 0*)

class piccolo.columns.defaults.**TimestampNow**

The current timestamp, in the local time of the machine that Python is running on.

20.8.4 Timestamptz

```
class piccolo.columns.defaults.TimestamptzCustom(year: int = 2000, month: int = 1, day: int = 1, hour:
    int = 0, minute: int = 0, second: int = 0,
    microsecond: int = 0)
```

```
class piccolo.columns.defaults.TimestamptzOffset(days: int = 0, hours: int = 0, minutes: int = 0,
    seconds: int = 0)
```

```
class piccolo.columns.defaults.TimestamptzNow
```

The current timestamp in UTC.

20.9 Testing

20.9.1 ModelBuilder

```
class piccolo.testing.model_builder.ModelBuilder
```

```
async classmethod build(table_class: type[TableInstance], defaults: dict[Column | str, Any] | None =
    None, persist: bool = True, minimal: bool = False) → TableInstance
```

Build a Table instance with random data and save async. If the Table has any foreign keys, then the related rows are also created automatically.

Parameters

- **table_class** – Table class to randomize.
- **defaults** – Any values specified here will be used instead of random values.
- **persist** – Whether to save the new instance in the database.
- **minimal** – If True then any columns with `null=True` are assigned a value of None.

Examples:

```
# Create a new instance with all random values:
manager = await ModelBuilder.build(Manager)

# Create a new instance, with certain defaults:
manager = await ModelBuilder.build(
    Manager,
    {Manager.name: 'Guido'}
)

# Create a new instance, but don't save it in the database:
manager = await ModelBuilder.build(Manager, persist=False)

# Create a new instance, with all null values set to None:
manager = await ModelBuilder.build(Manager, minimal=True)

# We can pass other table instances in as default values:
band = await ModelBuilder.build(Band, {Band.manager: manager})
```

```
classmethod build_sync(table_class: type[TableInstance], defaults: dict[Column | str, Any] | None =
    None, persist: bool = True, minimal: bool = False) → TableInstance
```

A sync wrapper around `build()`.

20.9.2 create_db_tables / drop_db_tables

`async piccolo.table.create_db_tables(*tables: type[Table], if_not_exists: bool = False) → None`

Creates the database table for each Table class passed in. The tables are created in the correct order, based on their foreign keys.

Parameters

- **tables** – The tables to create in the database.
- **if_not_exists** – No errors will be raised if any of the tables already exist in the database.

`piccolo.table.create_db_tables_sync(*tables: type[Table], if_not_exists: bool = False) → None`

A sync wrapper around `create_db_tables()`.

`async piccolo.table.drop_db_tables(*tables: type[Table]) → None`

Drops the database table for each Table class passed in. The tables are dropped in the correct order, based on their foreign keys.

Parameters

- **tables** – The tables to delete from the database.

`piccolo.table.drop_db_tables_sync(*tables: type[Table]) → None`

A sync wrapper around `drop_db_tables()`.

20.10 QueryString

`class piccolo.querystring.QueryString(template: str, *args: Any, query_type: str = 'generic', table: type[Table] | None = None, alias: str | None = None)`

When we're composing complex queries, we're combining QueryStrings, rather than concatenating strings directly. The reason for this is QueryStrings keep the parameters separate, so we can pass parameterised queries to the engine - which helps prevent SQL Injection attacks.

Parameters

- **template** – The SQL query, with curly brackets as placeholders for any values:

```
"WHERE {} = {}"
```

- **args** – The values to insert (one value is needed for each set of curly braces in the template).
 - **query_type** – The query type is sometimes used by the engine to modify how the query is run. For example, INSERT queries on old SQLite versions.
 - **table** – Sometimes the `piccolo.engine.base.Engine` needs access to the table that the query is being run on.
-

20.11 Custom Types

20.11.1 BasicTypes

`piccolo.custom_types.BasicTypes`

alias of `bytes` | `date` | `datetime` | `time` | `timedelta` | `Decimal` | `dict` | `float` | `int` | `list` | `str` | `UUID`

ABOUT PICCOLO V1

20th October

Piccolo v1 is now available!

We migrated to Pydantic v2, and also migrated Piccolo Admin to Vue 3, which puts the project in a good place moving forward.

We don't anticipate any major issues for people who are upgrading. If you encounter any bugs let us know.

Make sure you have v1 of Piccolo, Piccolo API, and Piccolo Admin.

2nd August 2023

Piccolo started in August 2018, and as of this writing is close to 5 years old.

During that time we've had very few, if any, breaking changes. Stability has always been very important, as we rely on it for our production apps.

So why release v1 now? We probably should have released v1 several years ago, but such are things. We now have some unavoidable breaking changes due to one of our main dependencies (Pydantic) releasing v2.

In v2, the core of Pydantic has been rewritten in Rust, and has impressive improvements in performance. Likewise, other libraries in the ecosystem (such as FastAPI) have moved to Pydantic v2. It only makes sense that Piccolo does it too.

In terms of your own code, you shouldn't see much difference. We removed the `pydantic_config_class` from `create_pydantic_model`, and replaced it with `pydantic_config`, but that's about it.

However, quite a bit of internal code in Piccolo and its sister libraries [Piccolo API](#) and [Piccolo Admin](#) had to be changed to support Pydantic v2. Supporting both Pydantic v1 and Pydantic v2 would be quite burdensome.

So Piccolo v1 will just use Pydantic v2 and above.

If you can't upgrade to Pydantic v2, then pin your Piccolo version to `0.118.0`. You can find the [docs here for 0.118.0](#).

TLDR

Install Piccolo:

```
pip install piccolo
```

Experiment with queries:

```
piccolo playground run
```

Give me an ASGI web app!

```
piccolo asgi new
```

FastAPI, Starlette, BlackSheep, Litestar, Ravyn, Lilya, Quart, Falcon and Sanic are currently supported, with more coming soon.

CHAPTER
TWENTYTHREE

VIDEOS

Piccolo has some [tutorial videos on YouTube](#), which are a great companion to the docs.

Symbols

`__getitem__()` (*piccolo.columns.column_types.Array* method), 93

A

`Abs` (*class in piccolo.query.functions.math*), 71

`add_m2m()` (*piccolo.table.Table* method), 245

`all()` (*piccolo.columns.column_types.Array* method), 93

`all_columns()` (*piccolo.table.Table* class method), 246

`all_related()` (*piccolo.table.Table* class method), 246

`alter()` (*piccolo.table.Table* class method), 247

`any()` (*piccolo.columns.column_types.Array* method), 93

`append()` (*piccolo.columns.column_types.Array* method), 94

`AppRegistry` (*class in piccolo.conf.apps*), 108

`Array` (*class in piccolo.columns.column_types*), 92

`ArrayAppend` (*class in piccolo.query.functions.array*), 66

`ArrayCat` (*class in piccolo.query.functions.array*), 66

`ArrayPrepend` (*class in piccolo.query.functions.array*), 66

`ArrayRemove` (*class in piccolo.query.functions.array*), 66

`ArrayReplace` (*class in piccolo.query.functions.array*), 67

`as_alias()` (*piccolo.columns.base.Column* method), 255

`AsyncTableTest` (*class in piccolo.testing.test_case*), 149

`AsyncTransactionTest` (*class in piccolo.testing.test_case*), 148

`AtTimeZone` (*class in piccolo.query.functions.datetime*), 68

`auto_update` (*piccolo.columns.base.ColumnKwargs* attribute), 257

`Avg` (*class in piccolo.query.functions.aggregate*), 64

B

`backwards()` (*in module piccolo.apps.migrations.commands.backwards*), 131

`BaseUser` (*class in piccolo.apps.user.tables*), 136

`BasicTypes` (*in module piccolo.custom_types*), 262

`BigInt` (*class in piccolo.columns.column_types*), 81

`BigSerial` (*class in piccolo.columns.column_types*), 82

`Boolean` (*class in piccolo.columns.column_types*), 78

`btree` (*piccolo.columns.indexes.IndexMethod* attribute), 260

`build()` (*piccolo.testing.model_builder.ModelBuilder* class method), 261

`build_sync()` (*piccolo.testing.model_builder.ModelBuilder* class method), 261

`Bytea` (*class in piccolo.columns.column_types*), 77

C

`cascade` (*piccolo.columns.OnDelete* attribute), 259

`cascade` (*piccolo.columns.OnUpdate* attribute), 259

`Cast` (*class in piccolo.query.functions.type_conversion*), 73

`cat()` (*piccolo.columns.column_types.Array* method), 93

`Ceil` (*class in piccolo.query.functions.math*), 71

`Char` (*class in piccolo.columns.column_types*), 85

`check()` (*in module piccolo.apps.migrations.commands.check*), 132

`choices` (*piccolo.columns.base.ColumnKwargs* attribute), 257

`Coalesce` (*class in piccolo.query.functions.conditional*), 67

`CockroachEngine` (*class in piccolo.engine.cockroach*), 121

`Column` (*class in piccolo.columns.base*), 254

`ColumnKwargs` (*class in piccolo.columns.base*), 257

`Concat` (*class in piccolo.query.functions.string*), 71

`Count` (*class in piccolo.query.functions.aggregate*), 64

`Count` (*class in piccolo.query.methods.select*), 257

`count()` (*piccolo.table.Table* class method), 247

`create_db_tables()` (*in module piccolo.table*), 262

`create_db_tables_sync()` (*in module piccolo.table*), 262

`create_index()` (*piccolo.table.Table* class method), 247

`create_pydantic_model()` (*in module piccolo.utils.pydantic*), 143

- create_schema() (*piccolo.schema.SchemaManager* method), 252
 create_table() (*piccolo.table.Table* class method), 248
 create_user() (*piccolo.apps.user.tables.BaseUser* class method), 136
 create_user_sync() (*piccolo.apps.user.tables.BaseUser* class method), 136
- ## D
- Date (class in *piccolo.columns.column_types*), 86
 DateOffset (class in *piccolo.columns.defaults*), 260
 Day() (in module *piccolo.query.functions.datetime*), 70
 db_column_name (*piccolo.columns.base.ColumnKwargs* attribute), 257
 ddl (*piccolo.columns.base.Column* property), 255
 deferred (*piccolo.engine.sqlite.TransactionType* attribute), 159
 delete() (*piccolo.table.Table* class method), 248
 DistinctOnError (class in *piccolo.query.mixins*), 51
 do_nothing (*piccolo.query.methods.insert.OnConflictAction* attribute), 58
 do_update (*piccolo.query.methods.insert.OnConflictAction* attribute), 58
 DoublePrecision (class in *piccolo.columns.column_types*), 82
 drop_db_tables() (in module *piccolo.table*), 262
 drop_db_tables_sync() (in module *piccolo.table*), 262
 drop_index() (*piccolo.table.Table* class method), 248
 drop_schema() (*piccolo.schema.SchemaManager* method), 253
- ## E
- Email (class in *piccolo.columns.column_types*), 86
 exclusive (*piccolo.engine.sqlite.TransactionType* attribute), 159
 exists() (*piccolo.table.Table* class method), 248
 Extract (class in *piccolo.query.functions.datetime*), 69
- ## F
- Floor (class in *piccolo.query.functions.math*), 71
 ForeignKey (class in *piccolo.columns.column_types*), 78
 forwards() (in module *piccolo.apps.migrations.commands.forwards*), 131
 freeze() (*piccolo.query.base.Query* method), 51
 from_dict() (*piccolo.table.Table* class method), 248
- ## G
- get_default_value() (*piccolo.columns.base.Column* method), 255
 get_m2m() (*piccolo.table.Table* method), 248
 get_readable() (*piccolo.table.Table* class method), 248
 get_related() (*piccolo.table.Table* method), 248
 get_select_string() (*piccolo.columns.base.Column* method), 255
 get_sql_value() (*piccolo.columns.base.Column* method), 255
 gin (*piccolo.columns.indexes.IndexMethod* attribute), 260
 gist (*piccolo.columns.indexes.IndexMethod* attribute), 260
- ## H
- hash (*piccolo.columns.indexes.IndexMethod* attribute), 260
 help_text (*piccolo.columns.base.ColumnKwargs* attribute), 257
 Hour() (in module *piccolo.query.functions.datetime*), 70
- ## I
- ilike() (*piccolo.columns.base.Column* method), 256
 immediate (*piccolo.engine.sqlite.TransactionType* attribute), 159
 index (*piccolo.columns.base.ColumnKwargs* attribute), 257
 index_method (*piccolo.columns.base.ColumnKwargs* attribute), 257
 indexes() (*piccolo.table.Table* class method), 248
 IndexMethod (class in *piccolo.columns.indexes*), 260
 insert() (*piccolo.table.Table* class method), 249
 Integer (class in *piccolo.columns.column_types*), 82
 Interval (class in *piccolo.columns.column_types*), 87
 is_not_null() (*piccolo.columns.base.Column* method), 256
 is_null() (*piccolo.columns.base.Column* method), 256
- ## J
- join_on() (*piccolo.columns.base.Column* method), 256
 JSON (class in *piccolo.columns.column_types*), 89
 JSONB (class in *piccolo.columns.column_types*), 89
- ## L
- LazyTableReference (class in *piccolo.columns*), 259
 Length (class in *piccolo.query.functions.string*), 72
 like() (*piccolo.columns.base.Column* method), 256
 list_schemas() (*piccolo.schema.SchemaManager* method), 253
 list_tables() (*piccolo.schema.SchemaManager* method), 253
 login() (*piccolo.apps.user.tables.BaseUser* class method), 136
 login_sync() (*piccolo.apps.user.tables.BaseUser* class method), 136

Lower (class in piccolo.query.functions.string), 72
Ltrim (class in piccolo.query.functions.string), 72

M

Max (class in piccolo.query.functions.aggregate), 65
Min (class in piccolo.query.functions.aggregate), 65
Minute() (in module piccolo.query.functions.datetime), 70
ModelBuilder (class in piccolo.testing.model_builder), 261
Month() (in module piccolo.query.functions.datetime), 70
move_table() (piccolo.schema.SchemaManager method), 253

N

no_action (piccolo.columns.OnDelete attribute), 259
no_action (piccolo.columns.OnUpdate attribute), 259
not_any() (piccolo.columns.column_types.Array method), 93
null (piccolo.columns.base.ColumnKwargs attribute), 257
NullIf (class in piccolo.query.functions.conditional), 68
Numeric (class in piccolo.columns.column_types), 82

O

objects() (piccolo.table.Table class method), 249
on_conflict() (piccolo.query.methods.insert.Insert method), 58
OnConflictAction (class in piccolo.query.methods.insert), 58
OnDelete (class in piccolo.columns), 259
OnUpdate (class in piccolo.columns), 259

P

PostgresEngine (class in piccolo.engine.postgres), 120
prepend() (piccolo.columns.column_types.Array method), 95
primary_key (piccolo.columns.base.ColumnKwargs attribute), 257

Q

QueryString (class in piccolo.querystring), 262
querystring (piccolo.table.Table property), 249

R

raw() (piccolo.table.Table class method), 249
Real (class in piccolo.columns.column_types), 83
ref() (piccolo.table.Table class method), 249
Refresh (class in piccolo.query.methods.refresh), 258
refresh() (piccolo.table.Table method), 250
remove() (piccolo.columns.column_types.Array method), 94

remove() (piccolo.table.Table method), 250
remove_m2m() (piccolo.table.Table method), 250
rename_schema() (piccolo.schema.SchemaManager method), 254
Replace (class in piccolo.query.functions.string), 72
replace() (piccolo.columns.column_types.Array method), 95
required (piccolo.columns.base.ColumnKwargs attribute), 257
restrict (piccolo.columns.OnDelete attribute), 259
restrict (piccolo.columns.OnUpdate attribute), 259
Reverse (class in piccolo.query.functions.string), 72
reverse() (piccolo.columns.column_types.ForeignKey method), 100
Round (class in piccolo.query.functions.math), 71
Rtrim (class in piccolo.query.functions.string), 72
run() (piccolo.query.methods.refresh.Refresh method), 258
run_sync() (piccolo.query.methods.refresh.Refresh method), 258

S

save() (piccolo.table.Table method), 250
SchemaManager (class in piccolo.schema), 252
Second() (in module piccolo.query.functions.datetime), 71
Secret (class in piccolo.columns.column_types), 85
secret (piccolo.columns.base.ColumnKwargs attribute), 257
select() (piccolo.table.Table class method), 250
Serial (class in piccolo.columns.column_types), 83
set_default (piccolo.columns.OnDelete attribute), 259
set_default (piccolo.columns.OnUpdate attribute), 259
set_null (piccolo.columns.OnDelete attribute), 259
set_null (piccolo.columns.OnUpdate attribute), 259
SmallInt (class in piccolo.columns.column_types), 84
SQLiteEngine (class in piccolo.engine.sqlite), 119
Strftime (class in piccolo.query.functions.datetime), 69
Sum (class in piccolo.query.functions.aggregate), 65

T

Table (class in piccolo.table), 245
table_exists() (piccolo.table.Table class method), 251
table_finder() (in module piccolo.conf.apps), 111
TableTest (class in piccolo.testing.test_case), 149
Text (class in piccolo.columns.column_types), 85
Time (class in piccolo.columns.column_types), 87
Timestamp (class in piccolo.columns.column_types), 88
TimestampCustom (class in piccolo.columns.defaults), 260
TimestampNow (class in piccolo.columns.defaults), 260

TimestampOffset (class in piccolo.columns.defaults),
260
Timestamptz (class in piccolo.columns.column_types),
88
TimestamptzCustom (class in piccolo.columns.defaults), 261
TimestamptzNow (class in piccolo.columns.defaults),
261
TimestamptzOffset (class in piccolo.columns.defaults), 261
to_dict() (piccolo.table.Table method), 251
TransactionType (class in piccolo.engine.sqlite), 159

U

unique (piccolo.columns.base.ColumnKwargs attribute),
257
update() (piccolo.table.Table class method), 251
update_password() (piccolo.apps.user.tables.BaseUser class method),
136
update_password_sync() (piccolo.apps.user.tables.BaseUser class method),
136
update_self() (piccolo.table.Table method), 252
Upper (class in piccolo.query.functions.string), 72
UUID (class in piccolo.columns.column_types), 84
UUID4 (class in piccolo.columns.defaults), 260
UUID7 (class in piccolo.columns.defaults), 260

V

value_type (piccolo.columns.base.Column attribute),
257
Varchar (class in piccolo.columns.column_types), 86

Y

Year() (in module piccolo.query.functions.datetime), 70