Piccolo Documentation

Release 0.5.1

Daniel Townsend

Sep 16, 2019

Contents:

CHAPTER 1

Getting Started

1.1 What is Piccolo?

Piccolo is a fast, easy to learn ORM.

Some of it's stand out features are:

- Support for sync and async - see *Sync and Async*.
- A builtin playground, which makes learning a breeze - see *Playground*.
- Works great with iPython and VSCode - see *Tab Completion*.
- Batteries included - a User model, JWT support, an admin, and more.

1.2 Installing Piccolo

1.2.1 Python

You need Python 3.7 or above installed on your system.

1.2.2 Pip

Now install piccolo, ideally inside a virtualenv:

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

# The important bit:
pip install piccolo
```

1.3 Playground

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

```
piccolo playground
```

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

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

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

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

1.3.1 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.

1.3.2 Advanced usage

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

1.3.3 Test queries

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

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

Give these queries a go:

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

1.3.4 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.

1.4 Installing Postgres

1.4.1 Mac

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

1.4.2 Ubuntu

On Ubuntu you can use apt.

1.4.3 Windows

For Windows, you can use a package manager like chocolatey.

1.4.4 Postgres version

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

1.4.5 What about other databases?

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

1.5 Sync and Async

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

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

1.5.1 Sync example

```
from my_schema import Band


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


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

1.5.2 Async example

```python
import asyncio
from my_schema import Band


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


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

1.5.3 Which to use?

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

Using the async version is useful for web applications which require high throughput, based on ASGI frameworks.

1.5.4 Explicit

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

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

1.6 Example Schema

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

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


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


class Band(Table):
    name = Varchar(max_length=100)
    manager = Varchar(max_length=100)
```

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

CHAPTER 2

Query Types

There are many different queries you can perform using Piccolo. For starters, focus on *select*, *insert* and *update*.

2.1 Select

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

To get all rows:

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

To get certain rows:

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

Or making an alias to make it shorter:

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

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

2.1.1 Joins

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

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

The joins can go several layers deep.

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

2.1.2 Query clauses

batch

See *batch*.

columns

By default all columns are returned from the queried table.

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

To restrict the returned columns, used the *columns* method.

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

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

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

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

first

See *first*.

limit

See *limit*.

order_by

See *order_by*.

output

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

To return the data as a JSON string:

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

where

See *where*.

2.2 Alter

This is used to modify an existing table.

Hint: It is typically used in conjunction with migrations - see *Migrations*.

2.2.1 add_column

Used to add a column to an existing table.

```
Band.alter().add_column(‘members’, Integer()).run_sync()
```

2.2.2 drop_column

Used to drop an existing column.

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

2.2.3 rename_column

Used to rename an existing column.

```
Band.alter().rename_column(Band.popularity, ‘rating’).run_sync()
```

2.2.4 set_null

Set whether a column is nullable or not.

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

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

2.2.5 set_unique

Used to change whether a column is unique or not.

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

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

2.3 Count

Returns the number of rows which match the query.

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

2.3.1 Query clauses

where

See *where*.

2.4 Create

This creates the table and columns in the database.

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

Alternatively, you can use `create_without_columns`, which just creates the table, without any columns.

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

Hint: It is typically used in conjunction with migrations - see *Migrations*.

2.5 Delete

This deletes any matching rows from the table.

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

2.5.1 Query clauses

where

See *where*

2.6 Drop

This removes a table and all its data from the database.

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

Hint: It is typically used in conjunction with migrations - see *Migrations*.

2.7 Exists

This checks whether any rows exist which match the criteria.

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

2.7.1 Query clauses

where

See *where*.

2.8 Insert

This is used to insert rows into the table.

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

We can insert multiple rows in one go:

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

2.8.1 add

You can also compose it as follows:

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

2.9 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 also useful to get objects back instead.

In Piccolo, an instance of a Table represents a row. Lets do an example.

```
# To get all objects:
Band.objects().run_sync()

# To get certain rows:
Band.objects().where(Band.name == 'Pythonistas').run_sync()

# Get the first row
Band.objects().first().run_sync()
```

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

2.9.1 Saving objects

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

```
# To get certain rows:
pythonistas = Band.objects().where(
    Band.name == 'Pythonistas'
).first().run_sync()

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

2.9.2 Deleting objects

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

```
# To get certain rows:
pythonistas = Band.objects().where(
    Band.name == 'Pythonistas'
).first().run_sync()

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

2.9.3 get_related

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

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

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

2.9.4 Query clauses

batch

See *batch*.

limit

See *limit*.

first

See *first*.

order_by

See *order_by*.

where

See *where* .

2.10 Raw

Should you need to, you can execute raw SQL.

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

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

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

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

2.11 Update

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

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

2.11.1 Query clauses

where

See *where*.

2.12 Transactions

Transactions allow multiple queries to be committed only once successful.

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

2.12.1 Usage

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

CHAPTER 3

Query Clauses

3.1 first

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

- *Objects*
- *Select*

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

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

Likewise, with objects:

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

3.2 limit

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

- *Objects*
- *Select*

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

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

Likewise, with objects:

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

3.3 order_by

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

- *Select*
- *Objects*

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

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

To order by descending:

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

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

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

3.4 where

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

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

It allows powerful filtering of your data.

3.4.1 Equal / Not Equal

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

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

3.4.2 Greater than / less than

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

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

3.4.3 like / ilike

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

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

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

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

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

3.4.4 not_like

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

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

3.4.5 is_in / not_in

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

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

3.4.6 Complex queries - and / or

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

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

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

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

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

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

```
b = Band

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

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

3.5 batch

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

- *Objects*
- *Select*

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

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

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

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

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

import asyncio
asyncio.run(get_batch())
```

CHAPTER 4

Schema

4.1 Defining a Schema

The first step is to define your schema. Create a file called `tables.py`.

This reflects the tables in your database. Each table consists of several columns.

```
"""
tables.py
"""
from piccolo.tables import Table
from piccolo.columns import Varchar


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

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

4.1.1 Connecting to the database

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

4.2 Column Types

Hint: You'll notice that all of the column names match their SQL equivalents.

4.2.1 Column

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

The following arguments apply to all column types:

default

```
class Band(Table):
    name = Varchar(default="hello")
```

null

Whether the column is nullable.

```
class Band(Table):
    name = Varchar(default="hello", null=False)
```

4.2.2 Varchar

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

4.2.3 ForeignKey

ForeignKey takes a *Table* argument.

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

4.2.4 Integer

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

4.2.5 Timestamp

```
class Band(Table):
    created = Timestamp()
```

4.2.6 Boolean

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

CHAPTER 5

Engines

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

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

5.1 Explicit

```
from piccolo.engine.sqlite import SQLiteEngine
from piccolo.tables 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()
```

5.2 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.

Here's an example config file:

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

```

(continues on next page)

(continued from previous page)

```

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

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

5.2.1 PICCOLO_CONF environment variable

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

In your terminal:

```
export PICCOLO_CONF=piccolo_conf_test
```

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

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

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

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


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

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

5.3 Engine types

5.3.1 SQLiteEngine

```
from piccolo.engine.sqlite import SQLiteEngine


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

5.3.2 PostgresEngine

```
from piccolo.engine.postgres import PostgresEngine


DB = PostgresEngine({
    'host': 'localhost',
```

(continues on next page)

(continued from previous page)

```
    'database': 'my_app',
    'user': 'postgres',
    'password': ''
})
```

5.4 Connection pool

Warning: This is currently only available for Postgres.

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

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

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


app = Starlette()


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


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

5.4.1 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_connnection_pool(max_size=20)
```

CHAPTER 6

Migrations

6.1 Creating migrations

Migrations are used to create the tables in the database.

```
piccolo new
```

This creates a migrations folder, along with a migration file.

The migration filename is a timestamp, which also serves as the migration ID.

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

The contents of the migration file look like this:

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

async def forwards():
    pass

async def backwards():
    pass
```

6.1.1 Populating the migration

At the moment, this migration does nothing when run - we need to populate the forwards and backwards functions.

```
from ..tables import Band

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

(continues on next page)

(continued from previous page)

```
async def forwards():
    transaction = Band._meta.db.transaction()

    transaction.add(
        Band.create_without_columns(),

        Band.alter().add_column(
            'name',
            Varchar(length=100)
        ),
    )

    await transaction.run()


async def backwards():
    await Band.drop().run()
```

6.1.2 Running migrations

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

```
piccolo forwards
```

Inspect your database, and a `band` table should now exist.

6.1.3 Reversing migrations

To reverse the migration, run this:

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

This executes the backwards function.

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

CHAPTER 7

Authentication

Piccolo ships with some basic authentication support out of the box.

7.1 BaseUser

Inherit from `BaseUser` to create your own User table.

```
from piccolo.extensions.user import BaseUser


class User(BaseUser, tablename="custom_user"):
    pass
```

Hint: A table name of *user* isn't allowed since it clashes with a keyword in Postgres - so override the `tablename`, if you choose to name your class `User`.

7.1.1 login

To login a user, do the following:

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

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

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

7.1.2 update_password / update_password_sync

To change a user's password:

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

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

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

CHAPTER 8

Features

8.1 Tab Completion

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

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

8.2 Supported Databases

8.2.1 Postgres

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

8.2.2 SQLite

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

8.3 Security

8.3.1 SQL Injection protection

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

8.4 Syntax

8.4.1 As close as possible to SQL

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

For example:

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

8.4.2 Get the SQL at any time

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

```
query = Band.select().columns(Band.name).where(Band.popularity >= 100)

print(query)
'SELECT name from band where popularity > 100'
```

CHAPTER 9

Playground

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

9.1 Advanced Playground Usage

9.1.1 Postgres

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

Install Postgres

See *Installing 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 --help` for details).

Connecting

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

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

CHAPTER 10

Ecosystem

10.1 Piccolo API

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

10.2 Piccolo Admin

Lets you create a powerful web GUI for your tables in two minutes. Read the docs.

10.3 Piccolo Scaffold

Allows you to quickly create a new REST API project, using the various Piccolo libraries and Starlette. Coming soon.

CHAPTER 11

Contributing

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

11.1 Get the tests running

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

11.2 Contributing to the docs

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

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

11.3 Code style

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

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

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

Type hints are used throughout the project.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search