SQL (Relational) Databases with Peewee

!!! warning If you are just starting, the tutorial SQL (Relational) Databases{.internal-link target=_blank} that uses SQLAlchemy should be enough.

```
Feel free to skip this.
```

If you are starting a project from scratch, you are probably better off with SQLAlchemy ORM (SQL (Relational) Databases{.internal-link target=_blank}), or any other async ORM.

If you already have a code base that uses Peewee ORM, you can check here how to use it with **FastAPI**.

!!! warning "Python 3.7+ required" You will need Python 3.7 or above to safely use Peewee with FastAPI.

Peewee for async

Peewee was not designed for async frameworks, or with them in mind.

Peewee has some heavy assumptions about its defaults and about how it should be used.

If you are developing an application with an older non-async framework, and can work with all its defaults, **it can be a great tool**.

But if you need to change some of the defaults, support more than one predefined database, work with an async framework (like FastAPI), etc, you will need to add quite some complex extra code to override those defaults.

Nevertheless, it's possible to do it, and here you'll see exactly what code you have to add to be able to use Peewee with FastAPI.

!!! note "Technical Details" You can read more about Peewee's stand about async in Python in the docs, an issue, a PR.

The same app

We are going to create the same application as in the SQLAlchemy tutorial (SQL (Relational) Databases{.internal-link target=_blank}).

Most of the code is actually the same.

So, we are going to focus only on the differences.

File structure

Let's say you have a directory named `my_super_project` that contains a sub-directory called `sql_app` with a structure like this:

```
.
└── sql_app
    ├── __init__.py
    ├── crud.py
    ├── database.py
    ├── main.py
    └── schemas.py
```

This is almost the same structure as we had for the SQLAlchemy tutorial.

Now let's see what each file/module does.

Create the Peewee parts

Let's refer to the file `sql_app/database.py`.

The standard Peewee code

Let's first check all the normal Peewee code, create a Peewee database:

```
{!../../../docs_src/sql_databases_peewee/sql_app/database.py!}
```

!!! tip Have in mind that if you wanted to use a different database, like PostgreSQL, you couldn't just change the string. You would need to use a different Peewee database class.

Note

The argument:

```
check_same_thread=False
```

is equivalent to the one in the SQLAlchemy tutorial:

```
connect_args={"check_same_thread": False}
```

...it is needed only for `SQLite`.

!!! info "Technical Details"

```
Exactly the same technical details as in [SQL (Relational) Databases](../tutorial/sql-databases.md#note){.internal-link target=_blank} apply.
```

Make Peewee async-compatible `PeeweeConnectionState`

The main issue with Peewee and FastAPI is that Peewee relies heavily on Python's `threading.local`, and it doesn't have a direct way to override it or let you handle connections/sessions directly (as is done in the SQLAlchemy tutorial).

And `threading.local` is not compatible with the new async features of modern Python.

!!! note "Technical Details" `threading.local` is used to have a "magic" variable that has a different value for each thread.

```
This was useful in older frameworks designed to have one single thread per request, no more, no less.

Using this, each request would have its own database connection/session, which is the actual final goal.

But FastAPI, using the new async features, could handle more than one request on the
```

```
same thread. And at the same time, for a single request, it could run multiple things
in different threads (in a threadpool), depending on if you use `async def` or normal
`def`. This is what gives all the performance improvements to FastAPI.
```

But Python 3.7 and above provide a more advanced alternative to `threading.local`, that can also be used in the places where `threading.local` would be used, but is compatible with the new async features.

We are going to use that. It's called `contextvars`.

We are going to override the internal parts of Peewee that use `threading.local` and replace them with `contextvars`, with the corresponding updates.

This might seem a bit complex (and it actually is), you don't really need to completely understand how it works to use it.

We will create a `PeeweeConnectionState`:

```
{!../../../docs_src/sql_databases_peewee/sql_app/database.py!}
```

This class inherits from a special internal class used by Peewee.

It has all the logic to make Peewee use `contextvars` instead of `threading.local`.

`contextvars` works a bit differently than `threading.local`. But the rest of Peewee's internal code assumes that this class works with `threading.local`.

So, we need to do some extra tricks to make it work as if it was just using `threading.local`. The `__init__`, `__setattr__`, and `__getattr__` implement all the required tricks for this to be used by Peewee without knowing that it is now compatible with FastAPI.

!!! tip This will just make Peewee behave correctly when used with FastAPI. Not randomly opening or closing connections that are being used, creating errors, etc.

```
But it doesn't give Peewee async super-powers. You should still use normal `def`
functions and not `async def`.
```

Use the custom `PeeweeConnectionState` class

Now, overwrite the `._state` internal attribute in the Peewee database `db` object using the new `PeeweeConnectionState`:

```
{!../../../docs_src/sql_databases_peewee/sql_app/database.py!}
```

!!! tip Make sure you overwrite `db._state` *after* creating `db`.

!!! tip You would do the same for any other Peewee database, including `PostgresqlDatabase`, `MySQLDatabase`, etc.

Create the database models

Let's now see the file `sql_app/models.py`.

Create Peewee models for our data

Now create the Peewee models (classes) for `User` and `Item`.

This is the same you would do if you followed the Peewee tutorial and updated the models to have the same data as in the SQLAlchemy tutorial.

!!! tip Peewee also uses the term "**model**" to refer to these classes and instances that interact with the database.

```
But Pydantic also uses the term "**model**" to refer to something different, the data validation, conversion, and documentation classes and instances.
```

Import `db` from `database` (the file `database.py` from above) and use it here.

```
{!../../../docs_src/sql_databases_peewee/sql_app/models.py!}
```

!!! tip Peewee creates several magic attributes.

```
It will automatically add an `id` attribute as an integer to be the primary key.

It will chose the name of the tables based on the class names.

For the `Item`, it will create an attribute `owner_id` with the integer ID of the `User`. But we don't declare it anywhere.
```

Create the Pydantic models

Now let's check the file `sql_app/schemas.py`.

!!! tip To avoid confusion between the Peewee *models* and the Pydantic *models*, we will have the file `models.py` with the Peewee models, and the file `schemas.py` with the Pydantic models.

```
These Pydantic models define more or less a "schema" (a valid data shape).

So this will help us avoiding confusion while using both.
```

Create the Pydantic *models* / schemas

Create all the same Pydantic models as in the SQLAlchemy tutorial:

```
{!../../../docs_src/sql_databases_peewee/sql_app/schemas.py!}
```

!!! tip Here we are creating the models with an `id`.

```
We didn't explicitly specify an `id` attribute in the Peewee models, but Peewee adds one automatically.

We are also adding the magic `owner_id` attribute to `Item`.
```

Create a `PeeweeGetterDict` for the Pydantic *models* / schemas

When you access a relationship in a Peewee object, like in `some_user.items`, Peewee doesn't provide a `list` of `Item`.

It provides a special custom object of class `ModelSelect`.

It's possible to create a `list` of its items with `list(some_user.items)`.

But the object itself is not a `list`. And it's also not an actual Python generator. Because of this, Pydantic doesn't know by default how to convert it to a `list` of Pydantic *models* / schemas.

But recent versions of Pydantic allow providing a custom class that inherits from `pydantic.utils.GetterDict`, to provide the functionality used when using the `orm_mode = True` to retrieve the values for ORM model attributes.

We are going to create a custom `PeeweeGetterDict` class and use it in all the same Pydantic *models* / schemas that use `orm_mode`:

```
{!../../../docs_src/sql_databases_peewee/sql_app/schemas.py!}
```

Here we are checking if the attribute that is being accessed (e.g. `.items` in `some_user.items`) is an instance of `peewee.ModelSelect`.

And if that's the case, just return a `list` with it.

And then we use it in the Pydantic *models* / schemas that use `orm_mode = True`, with the configuration variable `getter_dict = PeeweeGetterDict`.

!!! tip We only need to create one `PeeweeGetterDict` class, and we can use it in all the Pydantic *models* / schemas.

CRUD utils

Now let's see the file `sql_app/crud.py`.

Create all the CRUD utils

Create all the same CRUD utils as in the SQLAlchemy tutorial, all the code is very similar:

```
{!../../../docs_src/sql_databases_peewee/sql_app/crud.py!}
```

There are some differences with the code for the SQLAlchemy tutorial.

We don't pass a `db` attribute around. Instead we use the models directly. This is because the `db` object is a global object, that includes all the connection logic. That's why we had to do all the `contextvars` updates above.

Aso, when returning several objects, like in `get_users`, we directly call `list`, like in:

```
list(models.User.select())
```

This is for the same reason that we had to create a custom `PeeweeGetterDict`. But by returning something that is already a `list` instead of the `peewee.ModelSelect` the `response_model` in the *path operation* with

`List[models.User]` (that we'll see later) will work correctly.

Main FastAPI app

And now in the file `sql_app/main.py` let's integrate and use all the other parts we created before.

Create the database tables

In a very simplistic way create the database tables:

```
{!../../../docs_src/sql_databases_peewee/sql_app/main.py!}
```

Create a dependency

Create a dependency that will connect the database right at the beginning of a request and disconnect it at the end:

```
{!../../../docs_src/sql_databases_peewee/sql_app/main.py!}
```

Here we have an empty `yield` because we are actually not using the database object directly.

It is connecting to the database and storing the connection data in an internal variable that is independent for each request (using the `contextvars` tricks from above).

Because the database connection is potentially I/O blocking, this dependency is created with a normal `def` function.

And then, in each *path operation function* that needs to access the database we add it as a dependency.

But we are not using the value given by this dependency (it actually doesn't give any value, as it has an empty `yield`). So, we don't add it to the *path operation function* but to the *path operation decorator* in the `dependencies` parameter:

```
{!../../../docs_src/sql_databases_peewee/sql_app/main.py!}
```

Context variable sub-dependency

For all the `contextvars` parts to work, we need to make sure we have an independent value in the `ContextVar` for each request that uses the database, and that value will be used as the database state (connection, transactions, etc) for the whole request.

For that, we need to create another `async` dependency `reset_db_state()` that is used as a sub-dependency in `get_db()`. It will set the value for the context variable (with just a default `dict`) that will be used as the database state for the whole request. And then the dependency `get_db()` will store in it the database state (connection, transactions, etc).

```
{!../../../docs_src/sql_databases_peewee/sql_app/main.py!}
```

For the **next request**, as we will reset that context variable again in the `async` dependency `reset_db_state()` and then create a new connection in the `get_db()` dependency, that new request will have its own database state (connection, transactions, etc).

!!! tip As FastAPI is an async framework, one request could start being processed, and before finishing, another request could be received and start processing as well, and it all could be processed in the same thread.

```
But context variables are aware of these async features, so, a Peewee database state
set in the `async` dependency `reset_db_state()` will keep its own data throughout the
entire request.

And at the same time, the other concurrent request will have its own database state
that will be independent for the whole request.
```

Peewee Proxy

If you are using a Peewee Proxy, the actual database is at `db.obj`.

So, you would reset it with:

```Python
async def reset_db_state():
    database.db.obj._state._state.set(db_state_default.copy())
    database.db.obj._state.reset()
```

Create your FastAPI *path operations*

Now, finally, here's the standard **FastAPI** *path operations* code.

```Python
{!../../../docs_src/sql_databases_peewee/sql_app/main.py!}
```

About `def` vs `async def`

The same as with SQLAlchemy, we are not doing something like:

```Python
user = await models.User.select().first()
```

...but instead we are using:

```Python
user = models.User.select().first()
```

So, again, we should declare the *path operation functions* and the dependency without `async def`, just with a normal `def`, as:

```Python
# Something goes here
def read_users(skip: int = 0, limit: int = 100):
    # Something goes here
```

Testing Peewee with async

This example includes an extra *path operation* that simulates a long processing request with `time.sleep(sleep_time)`.

It will have the database connection open at the beginning and will just wait some seconds before replying back. And each new request will wait one second less.

This will easily let you test that your app with Peewee and FastAPI is behaving correctly with all the stuff about threads.

If you want to check how Peewee would break your app if used without modification, go the the `sql_app/database.py` file and comment the line:

```
# db._state = PeeweeConnectionState()
```

And in the file `sql_app/main.py` file, comment the body of the `async` dependency `reset_db_state()` and replace it with a `pass`:

```
async def reset_db_state():
#     database.db._state._state.set(db_state_default.copy())
#     database.db._state.reset()
    pass
```

Then run your app with Uvicorn:

```
$ uvicorn sql_app.main:app --reload

<span style="color: green;">INFO</span>:     Uvicorn running on
http://127.0.0.1:8000 (Press CTRL+C to quit)
```

Open your browser at http://127.0.0.1:8000/docs and create a couple of users.

Then open 10 tabs at http://127.0.0.1:8000/docs#/default/read_slow_users_slowusers__get at the same time.

Go to the *path operation* "Get `/slowusers/` " in all of the tabs. Use the "Try it out" button and execute the request in each tab, one right after the other.

The tabs will wait for a bit and then some of them will show `Internal Server Error`.

What happens

The first tab will make your app create a connection to the database and wait for some seconds before replying back and closing the database connection.

Then, for the request in the next tab, your app will wait for one second less, and so on.

This means that it will end up finishing some of the last tabs' requests earlier than some of the previous ones.

Then one the last requests that wait less seconds will try to open a database connection, but as one of those previous requests for the other tabs will probably be handled in the same thread as the first one, it will have the same database connection that is already open, and Peewee will throw an error and you will see it in the terminal, and the response will have an `Internal Server Error`.

This will probably happen for more than one of those tabs.

If you had multiple clients talking to your app exactly at the same time, this is what could happen.

And as your app starts to handle more and more clients at the same time, the waiting time in a single request needs to be shorter and shorter to trigger the error.

Fix Peewee with FastAPI

Now go back to the file `sql_app/database.py`, and uncomment the line:

```
db._state = PeeweeConnectionState()
```

And in the file `sql_app/main.py` file, uncomment the body of the `async` dependency `reset_db_state()`:

```
async def reset_db_state():
    database.db._state._state.set(db_state_default.copy())
    database.db._state.reset()
```

Terminate your running app and start it again.

Repeat the same process with the 10 tabs. This time all of them will wait and you will get all the results without errors.

...You fixed it!

Review all the files

Remember you should have a directory named `my_super_project` (or however you want) that contains a sub-directory called `sql_app`.

`sql_app` should have the following files:

- `sql_app/__init__.py`: is an empty file.
- `sql_app/database.py`:

```
{!../../../docs_src/sql_databases_peewee/sql_app/database.py!}
```

- `sql_app/models.py`:

```
{!../../../docs_src/sql_databases_peewee/sql_app/models.py!}
```

- `sql_app/schemas.py`:

```
{!../../../docs_src/sql_databases_peewee/sql_app/schemas.py!}
```

- `sql_app/crud.py`:

```
{!../../../docs_src/sql_databases_peewee/sql_app/crud.py!}
```

- `sql_app/main.py`:

```
{!../../../docs_src/sql_databases_peewee/sql_app/main.py!}
```

Technical Details

!!! warning These are very technical details that you probably don't need.

The problem

Peewee uses `threading.local` by default to store it's database "state" data (connection, transactions, etc).

`threading.local` creates a value exclusive to the current thread, but an async framework would run all the code (e.g. for each request) in the same thread, and possibly not in order.

On top of that, an async framework could run some sync code in a threadpool (using `asyncio.run_in_executor`), but belonging to the same request.

This means that, with Peewee's current implementation, multiple tasks could be using the same `threading.local` variable and end up sharing the same connection and data (that they shouldn't), and at the same time, if they execute sync I/O-blocking code in a threadpool (as with normal `def` functions in FastAPI, in *path operations* and dependencies), that code won't have access to the database state variables, even while it's part of the same request and it should be able to get access to the same database state.

Context variables

Python 3.7 has `contextvars` that can create a local variable very similar to `threading.local`, but also supporting these async features.

There are several things to have in mind.

The `ContextVar` has to be created at the top of the module, like:

```
some_var = ContextVar("some_var", default="default value")
```

To set a value used in the current "context" (e.g. for the current request) use:

```
some_var.set("new value")
```

To get a value anywhere inside of the context (e.g. in any part handling the current request) use:

```
some_var.get()
```

Set context variables in the `async` dependency `reset_db_state()`

If some part of the async code sets the value with `some_var.set("updated in function")` (e.g. like the `async` dependency), the rest of the code in it and the code that goes after (including code inside of `async` functions called with `await`) will see that new value.

So, in our case, if we set the Peewee state variable (with a default `dict`) in the `async` dependency, all the rest of the internal code in our app will see this value and will be able to reuse it for the whole request.

And the context variable would be set again for the next request, even if they are concurrent.

Set database state in the dependency `get_db()`

As `get_db()` is a normal `def` function, **FastAPI** will make it run in a threadpool, with a *copy* of the "context", holding the same value for the context variable (the `dict` with the reset database state). Then it can add database state to that `dict`, like the connection, etc.

But if the value of the context variable (the default `dict`) was set in that normal `def` function, it would create a new value that would stay only in that thread of the threadpool, and the rest of the code (like the *path operation functions*) wouldn't have access to it. In `get_db()` we can only set values in the `dict`, but not the entire `dict` itself.

So, we need to have the `async` dependency `reset_db_state()` to set the `dict` in the context variable. That way, all the code has access to the same `dict` for the database state for a single request.

Connect and disconnect in the dependency `get_db()`

Then the next question would be, why not just connect and disconnect the database in the `async` dependency itself, instead of in `get_db()`?

The `async` dependency has to be `async` for the context variable to be preserved for the rest of the request, but creating and closing the database connection is potentially blocking, so it could degrade performance if it was there.

So we also need the normal `def` dependency `get_db()`.