tornado-sqlalchemy Documentation

Release 0.6.1

Siddhant Goel

May 07, 2019

Contents

tornado-sqlalchemy is a Python library aimed at providing a set of helpers for using the SQLAlchemy database toolkit in tornado web applications.

CHAPTER 1

Installation

```
$ pip install tornado-sqlalchemy
```

CHAPTER 2

Background

Tornado is an asynchronous web framework, meaning that it allows you to handle multiple web requests in parallel, and in case one request is waiting for a certain I/O operation to finish, Tornado would continue processing the second request.

Getting ORMs to work with such a framework can be a little tricky. The author of SQLAlchemy explains this very nicely on StackOverflow.

The TL;DR version is that since ORMs allow you to define relationships between your database models (for example using foreign keys), you can never be sure which property-access or function call would make a database round-trip.

So, given that,

1. this contradiction exists and we can't do anything about it,
2. and that Tornado applications sometimes **do** end up doing database access,

the aim of this project is to provide a few helper functions which you can use to handle SQLAlchemy queries in your Tornado project, without adding another layer of abstraction.

A prerequisite is a understanding of the following things -

1. ioloop,
2. session handling, and,
3. connection and engine objects

CHAPTER 3

Why?

This library handles the following problems/use-cases -

- **Boilerplate** - Tornado does not bundle code to handle database connections. That's fine, because it's not in the business of writing database code anyway. Everyone ends up writing their own code - code to establish database connections, initialize engines, get/teardown sessions, and so on.
- **Asynchronous query execution** - ORMs are poorly suited for explicit asynchronous programming. You don't know what property access or what method call would end up hitting the database. For a situation like this, it's a good idea to decide on *what exactly* you want to execute in the background.
- **Database migrations** - Since you're using SQLAlchemy, you're probably also using alembic for database migrations. This again brings us to the point about boilerplate. If you're currently using SQLAlchemy with Tornado and have migrations setup using alembic, you likely have custom code written somewhere.

The intention here is to have answers to all three of these in a standardized library which can act as a central place for all the bugs features, and hopefully can establish best practices.

CHAPTER 4

Usage

Construct a `session_factory` using `make_session_factory` and pass it to your `Application` object.

```
>>> from tornado.web import Application
>>> from tornado_sqlalchemy import make_session_factory
>>>
>>> factory = make_session_factory(database_url)
>>> my_app = Application(handlers, session_factory=factory)
```

Add the `SessionMixin` to your request handlers, which makes the `make_session` function available in the GET/POST/... methods you're defining. Additionally, it also provides a `self.session` property, which (lazily) constructs and returns a new session object (which will be closed in the `on_finish` Tornado entry point).

To run database queries in the background, use the `as_future` function to wrap the SQLAlchemy Query into a Future object, which you can `await` on or `yield` to get the result.

```
>>> from tornado.gen import coroutine
>>> from tornado_sqlalchemy import SessionMixin, as_future
>>>
>>> class OldCoroutineRequestHandler(SessionMixin, RequestHandler):
...     @coroutine
...     def get(self):
...         with self.make_session() as session:
...             count = yield as_future(session.query(User).count)
...
...         self.write('{} users so far!'.format(count))
...
>>> class NativeCoroutineRequestHandler(SessionMixin, RequestHandler):
...     async def get(self):
...         with self.make_session() as session:
...             count = await as_future(session.query(User).count)
...
...         self.write('{} users so far!'.format(count))
```

To setup database migrations, make sure that your SQLAlchemy models are inheriting using the result from the provided `declarative_base`.

```
>>> from sqlalchemy import Column, BigInteger, String
>>> from tornado_sqlalchemy import declarative_base
>>>
>>> DeclarativeBase = declarative_base()
>>>
>>> class User(DeclarativeBase):
>>>     id = Column(BigInteger, primary_key=True)
>>>     username = Column(String(255), unique=True)
```

And use the same `DeclarativeBase` object in the `env.py` file that alembic is using.

For a complete usage example, refer to the examples/tornado_web.py.