tornado-sqlalchemy Documentation

Release 0.1.0

Siddhant Goel

Nov 02, 2017

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 slightly different from the rest of the web frameworks, in that it allows handling web requests asynchronously out of the box. At the same time, making database operations asynchronous (especially when you put an ORM in the picture) is not that straight forward.

Hence, the aim of this project is to provide a few helper functions which can help you handle your database operations easily in case you're combining the two libraries.

Before using this project, it's important to understand the main guideline this project has - **We assume that the user knows how to use the two frameworks.**

Tornado is not like any other web framework, and to make use of the asynchronous functions it provides, it's necessary to understand how it really behaves underneath. In other words, you should **know** how that ioloop works.

Similarly, SQLAlchemy is an amazing framework, but I cannot stress how important it is to understand how session handling works and how to work with connection and engine objects.

We are not trying to add another layer of abstraction. The only thing we're trying to do is provide a set of helper functions for applications that happen to use both Tornado and SQLAlchemy.

CHAPTER 3

Why?

It seems like we should first answer the question - why does this library exist in the first place? What problems/use-cases is it tackling?

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

Since we're talking about `yield`, please note that currently we don't support the native `asyncio.Future` objects included in Python 3. So if your request handlers are `async def`-ed, then calling `await` on the `future` that `as_future` returns would likely not work. But the good news is that a later version of this library should support this use case as well.

```
>>> from tornado.gen import coroutine
>>> from tornado_sqlalchemy import SessionMixin, as_future
>>>
>>> class MyRequestHandler(RequestHandler, SessionMixin):
...     @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))
```

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

```
>>> 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.