Flask SQLAlchemy

Quik Reference

Ferreira Juan David

5 de febrero de 2021

Resumen

Flask-SQLAlchemy es una extensión para Flask que agrega soporte para SQLAlchemy a nuestras aplicaciones. Su objetivo es simplificar el uso de SQLAlchemy con Flask proporcionando valores predeterminados útiles y helpers adicionales que facilitan la realización de tareas comunes.

Consulte la documentación de SQLAlchemy para aprender a trabajar con el ORM en profundidad. La siguiente documentación es una breve descripción general de las tareas más comunes, así como de las características específicas de Flask-SQLAlchemy.

Requisitos

Our Version	Python	Flask	SQLAlchemy
2.x	2.7, 3.4+	0.12+	0.8+ or 1.0.10+ w/Python 3.7
3.0+ (in dev)	2.7, 3.5+	1.0+	1.0+

Guía de Usuario

Inicio rápido

Flask-SQLAlchemy es fácil de usar en aplicaciones básicas y se escala fácilmente para aplicaciones más grandes. Para obtener la guía completa, consulte la documentación de la API en la clase SQLAlchemy.

Una vez instanciado, este objeto contiene todas las "funciones" y "helpers" de sqlalchemy y sqlalchemy.orm. Además, proporciona una clase llamada Model, que es una base declarativa que se puede usar para declarar modelos:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

    def __repr__(self):
        return '<User %r>' % self.username
```

Para crear la base de datos, importamos el objeto db desde un shell de Python o de Flask

```
(venv) D:\Documents\venv\Flask\sqlalchemy> flask shell
```

y ejecutamos el método SQLAlchemy.create_all() para crear la base de datos y sus tablas:

```
>>> from app import db
>>> db.create_all()
```

Lo que crea nuestra base de datos. Ahora para crear algunos usuarios

```
>>> from app import User
>>> admin = User(username='admin', email='admin@xyz.com')
>>> guest = User(username='guest', email='guest@xyz.com')
```

Pero aún no están en la base de datos, así que aseguré-monos de que estén:

```
>>> db.session.add(admin)
>>> db.session.add(guest)
>>> db.session.commit()
```

Acceder a los datos en la base de datos es muy fácil:

```
>>> User.query.all()
[<User u'admin'>, <User u'guest'>]
>>> User.query.filter_by(username='admin').first()
<User u'admin'>
```

Observemos que nunca hemos definido un método __init__ en la clase User. Esto es porque SQLAlchemy agrega un constructor implícito a todas las clases del modelo que acepta argumentos de palabras clave para todas sus columnas y relaciones. Si decidimos anular el constructor por cualquier motivo, debemos asegurarnos de seguir aceptando **kwargs y llamar al superconstructor con esos **kwargs para preservar este comportamiento:

```
class Foo(db.Model):
    # ...
    def __init__(self, **kwargs):
        super(Foo, self).__init__(**kwargs)
        # do custom stuff
```

Relaciones simples

SQLAlchemy se conecta a bases de datos relacionales y estas son realmente buenas para establecer relaciones. Como tal, tendremos un ejemplo de una aplicación que usa dos tablas que tienen una relación entre sí:

```
from datetime import datetime

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80), nullable=False)
    body = db.Column(db.Text, nullable=False)
    pub_date = db.Column(db.DateTime, nullable=False,
        default=datetime.utcnow)

    category_id = db.Column(db.Integer, db.ForeignKey('category.id'),
        nullable=False)
    category = db.relationship('Category',
        backref=db.backref('posts', lazy=True))

    def __repr__(self):
        return '<Post %r>' % self.title


class Category(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)

    def __repr__(self):
        return '<Category %r>' % self.name
```

Primero creemos algunos objetos:

```
>>> py = Category(name='Python')
>>> Post(title='Hello Python!', body='Python is pretty cool', category=py)
>>> p = Post(title='Snakes', body='Ssssssss')
>>> py.posts.append(p)
>>> db.session.add(py)
```

Como podemos ver, no es necesario agregar los objetos `Post` a la sesión. Dado que `Category` es parte de la sesión, también se agregarán todos los objetos asociados a él a través de relaciones. No importa si `db.session.add()` se llama antes o después de crear estos objetos. La asociación también se puede hacer en cualquier lado de la relación, por lo que se puede crear una publicación con una categoría o se puede agregar a la lista de publicaciones de la categoría.
Veamos las publicaciones. Acceder a ellos los cargará desde la base de datos, ya que la relación se carga de forma diferida, pero probablemente no notaremos la diferencia: cargar una lista es bastante rápido:

```
>>> py.posts
[<Post 'Hello Python!'>, <Post 'Snakes'>]
```

Si bien la carga diferida de una relación es rápida, puede convertirse fácilmente en un cuello de botella importante cuando se generan consultas adicionales en un bucle para más de unos pocos objetos. Para este caso, `SQLAlchemy` nos permite anular la estrategia de carga en el nivel de la consulta. Si deseamos que una sola consulta cargue todas las categorías y sus publicaciones, podemos hacerlo así:

```
>>> from sqlalchemy.orm import joinedload
>>> query = Category.query.options(joinedload('posts'))
>>> for category in query:
...     print category, category.posts
<Category u'Python'> [<Post u'Hello Python!'>, <Post u'Snakes'>]
```

Si deseamos obtener un objeto de consulta para esa relación, podemos hacerlo usando `with_parent()`. Excluyamos esa publicación sobre `Snakes`, por ejemplo:

```
>>> Post.query.with_parent(py).filter(Post.title != 'Snakes').all()
[<Post 'Hello Python!'>]
```

Camino a la iluminación

Lo único que necesita saber con respecto a `SQLAlchemy` es que:

1. `SQLAlchemy` le da acceso a las siguientes cosas:
   - Todas las funciones y clases de `sqlalchemy` y `sqlalchemy.orm`.
   - Una sesión de ámbito preconfigurada llamada `session`.
   - La `metadata` y la `engine`.
   - Los métodos `SQLAlchemy.create_all()` y `SQLAlchemy.drop_all()` para crear y eliminar tablas según los modelos.
   - Una clase `Model` base que es una base declarativa configurada.
2. La clase `Model` base declarativa se comporta como una clase `Python` normal, pero tiene un atributo `query` adjunto que se puede usar para consultar el modelo. (`Model` y `BaseQuery`)
3. Tiene que confirmar la sesión, pero no tiene que eliminarla al final de la solicitud, `Flask-SQLAlchemy` lo hace por usted