Flask SQLAlchemy

Quik Reference

Ferreira Juan David

5 de febrero de 2021

Resumen

A partir de `Flask-SQLAlchemy 0.12` puede conectarse fácilmente a múltiples bases de datos. Para lograrlo, preconfigura `SQLAlchemy` para que admita múltiples "enlaces".

¿Qué son las ataduras?. En `SQLAlchemy`, un enlace es algo que puede ejecutar sentencias `SQL` y generalmente es una conexión o un motor. En `Flask-SQLAlchemy`, los enlaces son siempre motores que se crean automáticamente entre bastidores. Luego, cada uno de estos motores se asocia con una clave corta (la clave de enlace). Esta clave se usa luego en el momento de la declaración del modelo para asociar un modelo con un motor específico.

Si no se especifica ninguna clave de enlace para un modelo, se utiliza la conexión predeterminada en su lugar (según la configuración de `SQLALCHEMY_DATABASE_URI`).

Configuración de ejemplo

La siguiente configuración declara tres conexiones de base de datos. El predeterminado especial, así como otros dos usuarios con nombre (para los usuarios) y `appmeta` (que se conecta a una base de datos `sqlite` para acceso de solo lectura a algunos datos que la aplicación proporciona internamente):

```
SQLALCHEMY_DATABASE_URI = 'postgres://localhost/main'
SQLALCHEMY_BINDS = {
    'users': 'mysqldb://localhost/users',
    'appmeta': 'sqlite:////path/to/appmeta.db'
}
```

Creación y eliminación de tablas

Los métodos `create_all()` y `drop_all()` de forma predeterminada operan en todos los enlaces declarados, incluido el predeterminado. Este comportamiento se puede personalizar proporcionando el parámetro de vinculación. Se necesita un solo nombre de enlace, `'__all__'` para hacer referencia a todos los enlaces o una lista de enlaces. El bind (`SQLALCHEMY_DATABASE_URI`) predeterminado se llama `None`:

```
>>> db.create_all()
>>> db.create_all(bind=['users'])
>>> db.create_all(bind='appmeta')
>>> db.drop_all(bind=None)
```

Refiriéndose a Binds

Si declara un modelo, puede especificar el enlace que se utilizará con el atributo `__bind_key__`:

```
class User(db.Model):
    __bind_key__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
```

Internamente, la clave de vinculación se almacena en el diccionario de información de la tabla como `'bind_key'`. Es importante saber esto porque cuando desee crear un objeto de tabla directamente, tendrá que ponerlo allí:

```
user_favorites = db.Table('user_favorites',
    db.Column('user_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('message_id', db.Integer, db.ForeignKey('message.id')),
    info={'bind_key': 'users'}
)
```

Si especificó `__bind_key__` en sus modelos, puede usarlos exactamente como está acostumbrado. El modelo se conecta a la propia conexión de base de datos especificada.

Soporte de señalización

Nos conectamos a las siguientes señales para recibir notificaciones antes y después de que se confirmen los cambios en la base de datos. Estos cambios solo se controlan si `SQLALCHEMY_TRACK_MODIFICATIONS` están habilitados en la configuración.

Nuevo en la versión 0.10.

Modificado en la versión 2.1: `before_models_committed` se activa correctamente.

En desuso desde la versión 2.1: esto estará deshabilitado de forma predeterminada en una versión futura.

`models_committed`**:**

Esta señal se envía cuando los modelos modificados se envían a la base de datos.

El remitente es la aplicación que emitió los cambios. Al receptor se le pasa el parámetro `changes` con una lista de tuplas en el formulario. (`model instance`, `operation`).

La operación es una de 'insert', 'update' y 'delete'.
before_models_committed:
Esta señal funciona exactamente igual models_committed pero se emite antes de que se lleve a cabo la confirmación.

Customizing

Flask-SQLAlchemy define valores predeterminados sensibles. Sin embargo, a veces se necesita personalización. Hay varias formas de personalizar cómo se definen e interactúan los modelos.
Estas personalizaciones se aplican en la creación del SQLAlchemy objeto y se extienden a todos los modelos derivados de su clase Model.

Clase Model

Todos los modelos de SQLAlchemy heredan de una clase base declarativa. Esto se expone como db.Model en Flask-SQLAlchemy, que amplían todos los modelos. Esto se puede personalizar subclasificando el predeterminado y pasando la clase personalizada a model_class.
El siguiente ejemplo le da a cada modelo una clave primaria entera o una clave externa para la herencia de tablas unidas.
Nota: Las claves primarias enteras para todo no son necesariamente el mejor diseño de base de datos (eso depende de los requisitos de su proyecto), esto es solo un ejemplo.

```
from flask_sqlalchemy import Model, SQLAlchemy
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declared_attr, has_inherited_table

class IdModel(Model):
    @declared_attr
    def id(cls):
        for base in cls.__mro__[1:-1]:
            if getattr(base, '__table__', None) is not None:
                type = sa.ForeignKey(base.id)
                break
        else:
            type = sa.Integer

        return sa.Column(type, primary_key=True)

db = SQLAlchemy(model_class=IdModel)

class User(db.Model):
    name = db.Column(db.String)

class Employee(User):
    title = db.Column(db.String)
```

Model Mixins

Si el comportamiento solo es necesario en algunos modelos en lugar de en todos los modelos, usamos las clases mixin para personalizar solo esos modelos. Por ejemplo, si algunos modelos deben realizar un seguimiento de cuándo se crean o actualizan:

```
from datetime import datetime

class TimestampMixin(object):
    created = db.Column(
        db.DateTime, nullable=False, default=datetime.utcnow)
    updated = db.Column(db.DateTime, onupdate=datetime.utcnow)

class Author(db.Model):
    ...

class Post(TimestampMixin, db.Model):
    ...
```

Clase Query

También es posible personalizar lo que está disponible para su uso en la propiedad especial query de los modelos. Por ejemplo, proporcionando un método get_or:

```
from flask_sqlalchemy import BaseQuery, SQLAlchemy

class GetOrQuery(BaseQuery):
    def get_or(self, ident, default=None):
        return self.get(ident) or default

db = SQLAlchemy(query_class=GetOrQuery)

# get a user by id, or return an anonymous user instance
user = User.query.get_or(user_id, anonymous_user)
```

Y ahora todas las consultas ejecutadas desde la propiedad especial query en los modelos Flask-SQLAlchemy podemos usar el método get_or como parte de sus consultas. Todas las relaciones definidas con db.relationship (pero no sqlalchemy.orm.relationship()) también se proporcionarán con esta funcionalidad.
También es posible definir una clase de consulta personalizada para relaciones individuales, proporcionando la palabra clave query_class en la definición. Esto funciona con ambos db.relationship y sqlalchemy.relationship:

```
class MyModel(db.Model):
    cousin = db.relationship('OtherModel', query_class=GetOrQuery)
```

Nota: Si una clase de consulta se define en una relación, tendrá prioridad sobre la clase de consulta adjunta a su modelo correspondiente.
También es posible definir una clase de consulta específica para modelos individuales anulando el atributo query_class de clase en el modelo:

```
class MyModel(db.Model):
    query_class = GetOrQuery
```

En este caso, el método get_or solo estará disponible en consultas que se originen en MyModel.query.