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`).

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, use 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 querypropiedad especial en los modelos `Flask-SQLAlchemy` pueden usar el `get_or`método 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`.

Metaclase del modelo

Advertencia: Las metaclases son un tema avanzado y probablemente no necesites personalizarlas para lograr lo que deseas. Se documenta principalmente aquí para mostrar cómo deshabilitar la generación de nombres de tablas.
La metaclase del modelo es responsable de configurar los componentes internos de `SQLAlchemy` al definir las subclases del modelo. `Flask-SQLAlchemy` agrega algunos comportamientos adicionales a través de mixins; su metaclase predeterminada `DefaultMeta`, los hereda todos.
BindMetaMixin: `__bind_key__` se extrae de la clase y se aplica a la tabla. Consulte Varias bases de datos con enlaces.

`NameMetaMixin`: Si el modelo no especifica una `__tablename__` clave principal pero sí especifica, se genera un nombre automáticamente.
Puede agregar sus propios comportamientos definiendo su propia metaclase y creando la base declarativa usted mismo. Asegúrese de seguir heredando de los mixins que desee (o simplemente herede de la metaclase predeterminada).
Pasar una clase base declarativa en lugar de una clase base de modelo simple, como se muestra arriba, `base_class` hará que `Flask-SQLAlchemy` use esta base en lugar de construir una con la metaclase predeterminada

```
from flask_sqlalchemy import SQLAlchemy
from flask_sqlalchemy.model import DefaultMeta, Model

class CustomMeta(DefaultMeta):
    def __init__(cls, name, bases, d):
        # custom class setup could go here

        # be sure to call super
        super(CustomMeta, cls).__init__(name, bases, d)

    # custom class-only methods could go here

db = SQLAlchemy(model_class=declarative_base(
    cls=Model, metaclass=CustomMeta, name='Model'))
```

También puede pasar cualquier otro argumento que desee `declarative_base()` para personalizar la clase base según sea necesario.

Deshabilitar la generación de nombres de tablas

Algunos proyectos prefieren configurar cada modelo `__tablename__` manualmente en lugar de depender de la detección y generación de `Flask-SQLAlchemy`. La generación de nombres de tablas se puede desactivar definiendo una `metaclase` personalizada.

```
from flask_sqlalchemy.model import BindMetaMixin, Model
from sqlalchemy.ext.declarative import DeclarativeMeta, declarative_base

class NoNameMeta(BindMetaMixin, DeclarativeMeta):
    pass

db = SQLAlchemy(model_class=declarative_base(
    cls=Model, metaclass=NoNameMeta, name='Model'))
```

Esto crea una base que aún admite la función `__bind_key__` pero no genera nombres de tablas.