Flask SQLAlchemy

Quik Reference

Ferreira Juan David

4 de febrero de 2021

Resumen

Generalmente, `Flask-SQLAlchemy` se comporta como una base declarativa configurada correctamente desde la extensión declarativa. Como tal, recomendamos leer los documentos de `SQLAlchemy` para obtener una referencia completa. Sin embargo, los casos de uso más comunes también se documentan aquí.

Cosas a tener en cuenta:

- La clase base para todos sus modelos se llama `db.Model`. Se almacena en la instancia de `SQLAlchemy` que debe crear. Consulte Inicio rápido para obtener más detalles.
- Algunas partes que se requieren en `SQLAlchemy` son opcionales en `Flask-SQLAlchemy`. Por ejemplo, el nombre de la tabla se establece automáticamente a menos que se anule. Se deriva del nombre de la clase convertido a minúsculas y con "`CamelCase`" convertido a "`camel_case`". Para anular el nombre de la tabla, establezca el atributo `__tablename__` de clase.

Declaración de modelos

```python
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
```

Usamos `Column` para definir una columna. El nombre de la columna es el nombre que le asigna. Si desea utilizar un nombre diferente en la tabla, puede proporcionar un primer argumento opcional que es una cadena con el nombre de columna deseado. Las claves primarias están marcadas con `primary_key=True`. Se pueden marcar varias claves como claves primarias, en cuyo caso se convierten en una clave primaria compuesta.
Los tipos de columna son el primer argumento de `Column`. Puede proporcionarlos directamente o llamarlos para especificarlos más (como proporcionar una longitud). Los siguientes tipos son los más comunes:

`Integer`: Un entero.

`String(size)`: Una cadena con una longitud máxima (opcional en algunas bases de datos, por ejemplo, `PostgreSQL`).

`Text`: Algo de texto Unicode más largo.

`DateTime`: Fecha y hora expresadas como objeto `datetime` de `Python`.

`Float`: Almacena valores de coma flotante.

`Boolean`: Almacena un valor booleano.

`PickleType`: Corregir

`LargeBinary`: Almacena grandes datos binarios arbitrarios.

Relaciones One-to-Many

Las relaciones más comunes son las relaciones de uno a varios. Debido a que las relaciones se declaran antes de que se establezcan, puede usar cadenas para hacer referencia a clases que aún no se han creado (por ejemplo, si `Person` define una relación a `Address` que se declara más adelante en el archivo).
Las relaciones se expresan con la función `relationship()`. Sin embargo, la clave *foránea* debe declararse por separado con la clase `ForeignKey`:

```python
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    addresses = db.relationship('Address', backref='person', lazy=True)

class Address(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), nullable=False)
    person_id = db.Column(db.Integer, db.ForeignKey('person.id'), nullable=False)
```

¿Qué `db.relationship()` utilizar?. Esa función devuelve una nueva propiedad que puede hacer varias cosas. En este caso, le dijimos que apunte a la clase `Address` y cargue varios de ellos. *¿Cómo sabemos que esto devolverá más de una dirección?*. Porque `SQLAlchemy` adivina un valor predeterminado útil de su declaración. Si queremos tener una relación uno-a-uno puede pasar el parámetro `uselist=False` a `relationship()`.

Dado que una persona sin nombre o una dirección de correo electrónico sin dirección asociada no tiene sentido, `nullable=False` le dice a `SQLAlchemy` que cree la columna como `NOT NULL`. Esto está implícito para las columnas de clave *primaria*, pero es una buena idea especificarlo para todas las demás columnas, para que dejemos en claro a otras personas que trabajan en nuestro código que realmente deseaba una columna anulable y no simplemente se olvidó de agregarla.
Entonces, ¿qué significan `backref` y `lazy`?. `backref` es una forma sencilla de declarar también una nueva propiedad en la clase `Address`. A continuación, también puede utilizar `my_address.person` para llegar a la persona en esa dirección. `lazy` define cuándo `SQLAlchemy` cargará los datos de la base de datos:

- `'select'/True` (que es el valor predeterminado, pero explícito es mejor que implícito) significa que `SQLAlchemy` cargará los datos según sea necesario de una sola vez utilizando una declaración de selección estándar.
- `'joined'/False` le dice a `SQLAlchemy` que cargue la relación en la misma consulta que el padre usando una declaración `JOIN`.
- `'subquery'` funciona como `'joined'` pero en su lugar `SQLAlchemy` usará una subconsulta.
- `'dynamic'es especial` y puede resultar útil si tiene muchos elementos y siempre desea aplicarles filtros `SQL` adicionales. En lugar de cargar los elementos, `SQLAlchemy` devolverá otro objeto de consulta que puede refinar aún más antes de cargar los elementos. Tenga en cuenta que esto no se puede convertir en una estrategia de carga diferente al realizar consultas, por lo que a menudo es una buena idea evitar usar esto a favor de `lazy=True`.Se puede crear un objeto de consulta equivalente a una relación dinámica `user.address` usando `Address.query.with_parent(user)` mientras se puede usar la carga diferida o ansiosa en la relación en sí, según sea necesario.

¿Cómo se define el `lazy status` para `backrefs`?. Usando la función `backref()`:

```
class Person(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    addresses = db.relationship('Address', lazy='select', backref=db.backref('person', lazy='joined'))
```

Relación muchos a muchos

Si desea utilizar relaciones de varios a varios, deberá definir una tabla auxiliar que se utilice para la relación. Para esta mesa auxiliar, se recomienda encarecidamente no utilizar un modelo, sino una mesa real:

```
tags = db.Table('tags',
    db.Column('tag_id', db.Integer, db.ForeignKey('tag.id'), primary_key=True),
    db.Column('page_id', db.Integer, db.ForeignKey('page.id'), primary_key=True)
)

class Page(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    tags = db.relationship('Tag', secondary=tags, lazy='subquery', backref=db.backref('pages', lazy=True)

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
```

Aquí configuramos `Page.tags` para que se cargue inmediatamente después de cargar una página, pero usando una consulta separada. Esto siempre da como resultado dos consultas al recuperar una página, pero al consultar varias páginas no obtendrá consultas adicionales.

La lista de páginas para una etiqueta, por otro lado, es algo que rara vez se necesita. Por ejemplo, no necesitará esa lista al recuperar las etiquetas de una página específica. Por lo tanto, el `backref` está configurado para cargarse de forma diferida, de modo que acceder a él por primera vez desencadenará una consulta para obtener la lista de páginas para esa etiqueta. Si necesita aplicar más opciones de consulta en esa lista, puede cambiar a la estrategia `'dynamic'`, con los inconvenientes mencionados anteriormente, u obtener un objeto de consulta `Page.query.with_parent(some_tag)` y luego usarlo exactamente como lo haría con el objeto de consulta de una relación dinámica.