Flask SQLAlchemy

Quik Reference

Ferreira Juan David

5 de febrero de 2021

Resumen

Si pensamos en utilizar una sola aplicación, podemos saltar este capítulo. Simplemente pasemos nuestra aplicación al constructor `SQLAlchemy` y estará listo. Sin embargo, si deseamos utilizar más de una aplicación o crear la aplicación dinámicamente en una función, debemos seguir leyendo.

API Reference: Configuración

`class flask_sqlalchemy.SQLAlchemy(app=None, use_native_unicode=True, session_options=None, metadata=None, query_class=<class 'flask_sqlalchemy.BaseQuery'>, model_class=<class 'flask_sqlalchemy.model.Model'>, engine_options=None)`
Esta clase se utiliza para controlar la integración de `SQLAlchemy` en una o más aplicaciones `Flask`. Dependiendo de cómo inicialice el objeto, se puede usar de inmediato o se adjuntará según sea necesario a una aplicación `Flask`.
Hay dos modos de uso que funcionan de manera muy similar.

La *primera* es vincular la instancia a una aplicación `Flask` muy específica:

```
app = Flask(__name__)
db = SQLAlchemy(app)
```

La *segunda* posibilidad es crear el objeto una vez y configurar la aplicación más tarde para admitirlo:

```
db = SQLAlchemy()

def create_app():
    app = Flask(__name__)
    db.init_app(app)
    return app
```

La diferencia entre los dos es que en el **primer caso** los métodos como `create_all()` y `drop_all()` funcionarán todo el tiempo, pero en el **segundo caso** `flask.Flask.app_context()` tiene que existir.
De forma predeterminada, `Flask-SQLAlchemy` aplicará algunas configuraciones específicas de backend para mejorar nuetra experiencia con ellos.
A partir de `SQLAlchemy 0.6`, `SQLAlchemy` probará la biblioteca en busca de compatibilidad nativa con `Unicode`. Si detecta `unicode`, dejará que la biblioteca se encargue de eso; de lo contrario, lo hará ella misma. A veces, esta detección puede fallar, en cuyo caso es posible que desee establecer `use_native_unicode` (o la clave de configuración `SQLALCHEMY_NATIVE_UNICODE`) en `False`. Tenga en cuenta que la clave de configuración anula el valor que le pasa al constructor. El soporte directo para `use_native_unicode` y `SQLALCHEMY_NATIVE_UNICODE` está obsoleto a partir de la `v2.4` y se eliminará en la `v3.0`. Podemos usar en su lugar `engine_options` y `SQLALCHEMY_ENGINE_OPTIONS` . Esta clase también nos proporcionará acceso a todas las funciones y clases de `SQLAlchemy` desde los módulos `sqlalchemy` y `sqlalchemy.orm`. Entonces podemos declarar modelos como este:

```
class User(db.Model):
    username = db.Column(db.String(80), unique=True)
    pw_hash = db.Column(db.String(80))
```

Aún podemos usar directamente `sqlalchemy` y `sqlalchemy.orm`, pero tengamos en cuenta que las personalizaciones de `Flask-SQLAlchemy` están disponibles solo a través de una instancia de esta clase `SQLAlchemy`. Las clases de consulta por default a `BaseQuery` para `db.Query` , `db.Model.query_class` y la clase de consulta predeterminada `query_class` para `db.relationship` y `db.backref`. Si utiliza estas interfaces a través `sqlalchemy` y `sqlalchemy.orm` directamente, la clase de consulta predeterminada será la de `sqlalchemy`.

Compruebe los tipos con cuidado No realice comprobaciones de tipo o es instanciado en *db.Table* , que emula el comportamiento de la tabla pero no es una clase. `db.Table` expone la interfaz `Table` , pero es una función que permite la omisión de metadatos.

El parámetro `session_option`, si se proporciona, es un diccionario de parámetros que se pasarán al constructor de la sesión. Consulte las opciones estándar de `Session`.
Nuevo en la versión 0.10: La función `session_options` fue añadida.
Nuevo en la versión 0.16: scopefunc ahora se acepta en `session_options` . Permite especificar una función personalizada que definirá el alcance de la sesión SQLAlchemy.
Nuevo en la versión 2.1: Se agregó el parámetro de metadatos . Esto permite establecer convenciones de nomenclatura personalizadas, entre otras cosas no triviales.
Se agregó el parámetro `query_class` , para permitir la personalización de la clase de consulta, en lugar del predeterminado de `BaseQuery`.
Se agregó el parámetro `model_class`, que permite usar una clase de modelo personalizada en lugar de `Model`.
Modificado en la versión 2.1: utilice la misma clase de consulta en la sesión, `Model.query` y `Query`.
Nuevo en la versión 2.4: La función `engine_options` fue añadida.
Modificado en la versión 2.4: el parámetro use_native_unicode quedó obsoleto.
Modificado en la versión 2.4.3: `COMMIT_ON_TEARDOWN` está en desuso y se eliminará en la `versión 3.1`. En su lugar, llame `db.session.commit()` directamente.

`Query = None` Clase de consulta predeterminada utilizada por `Model.query` y otras consultas. Personaliza esto pasando `query_class` a `SQLAlchemy()`. Por defecto es `BaseQuery`.

`apply_driver_hacks(app, sa_url, options)`: Este método se llama antes de la creación del motor y se utiliza para inyectar trucos específicos del controlador en las opciones. El parámetro de opciones es un diccionario de argumentos de palabras clave que luego se usará para llamar a la función `sqlalchemy.create_engine()`.

La implementación predeterminada proporciona algunos valores predeterminados más cuerdos para cosas como tamaños de grupo para `MySQL` y `sqlite`. También inyecta la configuración de `SQLALCHEMY_NATIVE_UNICODE`.

`create_all( bind = '__ all__' , app = None )`: Crea todas las tablas.

Modificado en la versión 0.12: se agregaron parámetros.

`create_engine( sa_url , engine_opts )` Anule este método para tener la última palabra sobre cómo se crea el motor `SQLAlchemy`.

En la mayoría de los casos, querrá usar la 'SQLALCHEMY_ENGINE_OPTIONS' variable de configuración o establecer `engine_options` para `SQLAlchemy()`.

`create_scoped_session( opciones = None )`: Cree un *scoped_session* en la factoría a partir de `create_session()`.

Se puede establecer una clave adicional 'scopefunc' en el diccionario `options` para especificar una función de alcance personalizada. Si no se proporciona, se utiliza la identidad de la pila de contexto de la aplicación de `Flask`. Esto asegurará que las sesiones se creen y eliminen con el ciclo de solicitud/respuesta, y debería estar bien en la mayoría de los casos.

Parámetros $\Longrightarrow$ opciones - *diccionario de argumentos de palabras claves pasado a la clase* `create_session`.

`create_session( options )` Cree el método de factoría de sesiones utilizada por `create_scoped_session()`.

El método de factoría debe devolver un objeto que SQLAlchemy reconoce como una sesión, o el registro de eventos de sesión puede generar una excepción.

Los métodos de factoría válidas incluyen una clase `Session` o una `sessionmaker`.

La implementación predeterminada creamos un `sessionmakerfor SignallingSession`.

Parámetros $\Longrightarrow$ opciones - *diccionario de argumentos de palabras claves pasado a la clase* `session`.

`drop_all(bind='__all__', app=None)`: Elimina todas las tablas.

Modificado en la versión 0.12: se agregaron parámetros.

`engine`: Da acceso al motor. Si la configuración de la base de datos está vinculada a una aplicación específica (inicializada con una aplicación), esto siempre devolverá una conexión a la base de datos. Sin embargo, si se utiliza la aplicación actual, esto podría generar un error `RuntimeError` si no hay ninguna aplicación activa en este momento.

`get_app(reference_app = None)`: Método auxiliar que implementa la lógica para buscar una aplicación.

`get_binds(app = None)`: Devuelve un diccionario con una tabla $\Longrightarrow$ mapeo del motor.

Esto es adecuado para el uso de `sessionmaker(binds = db.get_binds(app))`.

`get_engine(app=None, bind=None)`: Devuelve un motor específico.