Seminario de Patrones - SQLAlchemy
Travelix

Daniel Toledo Daniel Machado Osvaldo Moreno José Antonio Concepción
C-311 - Agencia de Viajes. Equipo 3 Semigrupo 2.

8 de marzo de 2024

Índice

1. SQLAlchemy

SQLAlchemy es una biblioteca que se utiliza para interactuar con una amplia variedad de bases de datos. Permite crear modelos de datos y consultas de una manera que se asemeja a las clases y declaraciones normales de Python. Creada por Mike Bayer en 2005, SQLAlchemy es utilizada por muchas empresas grandes y pequeñas, y muchos la consideran la forma de facto de trabajar con bases de datos relacionales en Python.

Se puede utilizar para conectarse a las bases de datos más comunes como Postgres, MySQL, SQLite, Oracle y muchas otras. También proporciona una forma de agregar soporte para otras bases de datos relacionales. Amazon Redshift, que utiliza un dialecto personalizado de PostgreSQL, es un excelente ejemplo de soporte de base de datos agregado por la comunidad.

El ORM de SQLAlchemy es similar a muchos otros ORM de otros lenguajes. Se centra en el modelo de dominio de la aplicación y aprovecha el patrón de unidad de trabajo para mantener el estado del objeto. También proporciona una abstracción de alto nivel sobre el lenguaje de expresión SQL que permite al usuario trabajar de una manera más idiomática. Se puede combinar el uso del ORM con el lenguaje de expresión SQL para crear aplicaciones muy potentes. El ORM aprovecha un sistema declarativo similar a los sistemas de registro activo utilizados por muchos otros ORMs, como el que se encuentra en Ruby on Rails.

Si bien el ORM es extremadamente útil, se debe tener en cuenta que existe una diferencia entre la forma en que las clases pueden relacionarse y cómo funcionan las relaciones subyacentes de la base de datos.

Algunas características clave de SQLAlchemy incluyen:

- **Mapeo de Objetos:** SQLAlchemy permite mapear clases de Python a tablas en la base de datos, proporcionando una interfaz orientada a objetos para trabajar con los datos. Esto significa que los desarrolladores pueden manipular datos utilizando objetos y métodos en lugar de escribir consultas SQL manualmente.
- **Expresiones SQL:** La biblioteca proporciona un lenguaje expresivo de construcción de consultas que permite a los desarrolladores definir consultas de manera programática utilizando Python. Esto facilita la creación de consultas complejas y dinámicas.
- **Transacciones y Manejo de Sesiones:** SQLAlchemy maneja de manera eficiente las transacciones y proporciona un sistema de sesiones que facilita la gestión de la conexión y el control de las transacciones en la base de datos.
- **Soporte para Diversos Sistemas de Bases de Datos:** SQLAlchemy es compatible con varios sistemas de gestión de bases de datos relacionales, como PostgreSQL, MySQL, SQLite y Oracle, lo que proporciona flexibilidad en la elección de la base de datos subyacente.
- **Soporte para Modelos de Datos Complejos:** Permite la definición de relaciones complejas entre las tablas de la base de datos, como relaciones uno a uno, uno a muchos y muchos a muchos.
- **Extensibilidad:** SQLAlchemy es altamente extensible, lo que significa que los desarrolladores pueden personalizar y ampliar su funcionalidad según sea necesario.

Gestores a los que se conecta y cómo lo hace:

SQLAlchemy se puede conectar a varios gestores de bases de datos relacionales, incluyendo PostgreSQL. La conexión se realiza a través de una cadena de conexión que proporciona información como el tipo de

base de datos, el usuario, la contraseña, la dirección del servidor y otros parámetros relevantes. Aquí hay un ejemplo de cadena de conexión para PostgreSQL:

```
from sqlalchemy import create_engine

# Cadena de conexion para PostgreSQL
db_url = "postgresql://user:password@localhost/nombre_de_la_base_de_datos"
engine = create_engine(db_url)
```

Consultas (simples, join, group_by) con PostgreSQL:

SQLAlchemy proporciona una interfaz expresiva para realizar consultas. Aquí hay algunos ejemplos:

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()

# Consulta simple
users = session.query(User).all()
```

```
from sqlalchemy.orm import joinedload

# Consulta con join
query = session.query(User).options(joinedload(User.address)).all()
```

```
from sqlalchemy import func

# Consulta con group_by
result = session.query(func.count(User.id), User.name).group_by(User.name).all()
```

Operaciones CRUD con persistencia en PostgreSQL:

SQLAlchemy facilita las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) a través de objetos de sesión. Aquí hay ejemplos:

```
# Crear un nuevo usuario
new_user = User(name='JohnDoe', age=25)
session.add(new_user)
session.commit()
```

```
# Consultar usuarios
users = session.query(User).all()
```

```
# Actualizar un usuario existente
user = session.query(User).filter_by(name='JohnDoe').first()
user.age = 26
session.commit()
```

```python
# Eliminar un usuario
user_to_delete = session.query(User).filter_by(name='JohnDoe').first()
session.delete(user_to_delete)
session.commit()
```

Estos son ejemplos básicos, y SQLAlchemy proporciona muchas más funciones y características para operaciones CRUD y consultas avanzadas.

Generación del ORM en SQLAlchemy

En SQLAlchemy, la generación del ORM es principalmente de tipo "code first". Esto significa que se definen los modelos de datos en código utilizando clases de Python y luego SQLAlchemy se encarga de traducir estos modelos a tablas en la base de datos.

Definición de Modelos:

```python
from sqlalchemy import Column, Integer, String, create_engine
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)
```

Creación de la Base de Datos:

Después de definir los modelos, se utiliza SQLAlchemy para crear la base de datos y las tablas asociadas. Se puede hacer llamando al método `create_all()` en la instancia del objeto `Base`.

```python
engine =
    create_engine('postgresql://user:password@localhost/nombre_de_la_base_de_datos')
Base.metadata.create_all(engine)
```

En este ejemplo, se utiliza una base de datos PostgreSQL.

Una vez que la base de datos y las tablas están creadas, es posible interactuar con ellas utilizando instancias de las clases de modelo.

Interacción con la Base de Datos

```python
# Crear una nueva instancia de User
new_user = User(name='JohnDoe', age=25)

# Agregar el nuevo usuario a la sesion y confirmar la transaccion
session.add(new_user)
session.commit()

# Consultar usuarios
users = session.query(User).all()
```

1.1. Patrones de Acceso a Datos

Los patrones de Acceso a Datos de SQLAlchemy son: **Identity Map**, **Unit of Work**, **Data Mapper**, **Repository** y **Lazy Load**.

Identity Map

¿En qué consiste?:

El patrón Identity Map asegura que una entidad se cargue solo una vez en la memoria del programa, independientemente de cuántas veces se haya solicitado desde diferentes partes del código.

¿Para qué lo usa el ORM?:

Evita cargar múltiples instancias de la misma entidad, manteniendo una única referencia para cada entidad en la memoria.

¿Cómo lo usa el ORM?:

SQLAlchemy implementa Identity Map automáticamente. Para esto usa el objeto Session, que es la interfaz para el uso del ORM, es decir la carga y persistencia de los datos. Una Session mantiene una referencia a las entidades mapeadas que están presentes en su contexto e implementa un patrón fachada para los patrones de Identity Map y el Unit of Work. El Identity Map mantiene un único mapeo de cada identidad por Session, eliminando los problemas de identidades duplicadas.

Cuando se consulta una entidad con un identificador específico, SQLAlchemy verifica si esa entidad ya está en la memoria. Si es así, devuelve la instancia existente en lugar de cargarla nuevamente desde la base de datos.

Activar/Desactivar:

El ORM controla el Identity Map internamente, y no brinda una opción directa para activarlo/desactivarlo manualmente.

Unit of Work

¿En qué consiste?:

El patrón Unit of Work encapsula un conjunto de operaciones de datos, y una lista de acciones que son necesarias para considerar exitosa la transacción, de esta manera se garantiza la consistencia. Es usado en problemas de concurrencia y estabilidad.

¿Para qué lo usa el ORM?:

Garantiza que las operaciones de inserción, actualización y eliminación se realicen en conjunto como una transacción única, evitando cambios parciales y asegurando la integridad de la base de datos.

¿Cómo lo usa el ORM?:

La clase Session implementa el patrón Unity of Work, para proveer un proceso de persistencia de los cambios de estados de la base de datos. Todas las operaciones de manipulación de datos se realizan en una sesión y se confirman como una transacción única.

Activar/Desactivar:

El ORM brinda la capacidad de activar y desactivar explícitamente las transacciones utilizando el método ‘commit()‘ para confirmar los cambios o ‘rollback()‘ para revertir los cambios en una sesión.

Data Mapper

¿En qué consiste?:

El patrón Data Mapper separa la lógica de negocio del código de acceso a la base de datos, asignando una clase independiente para mapear los objetos de dominio a las tablas de la base de datos.

¿Para qué lo usa el ORM?:

Facilita el mapeo entre objetos de dominio y la estructura de la base de datos sin que la lógica de la base de datos afecte directamente a los objetos de dominio.

¿Cómo lo usa el ORM?:

SQLAlchemy implementa un Data Mapper a través de las clases de modelos de SQLAlchemy. Cada clase de modelo representa una entidad de la base de datos y proporciona un mapeo entre la estructura de la base de datos y los objetos de dominio. A través de mapper() y Table se pueden lograr la utilización del patrón.

Activar/Desactivar:

En SQLAlchemy, el uso del Data Mapper está intrínsecamente integrado en la definición de clases de modelos. No hay una opción directa para activar/desactivar esta funcionalidad de forma independiente.

Repository

¿En qué consiste?:

El patrón Repository media entre las capas de dominio y mapeo de datos utilizando una interfaz similar a una colección para acceder a los objetos de dominio.

¿Para qué lo usa el ORM?:

El patrón Repository es una abstracción que se utiliza para encapsular la lógica de acceso a datos. Su objetivo es proporcionar una interfaz coherente para trabajar con objetos del dominio sin preocuparse por los detalles específicos de la base de datos. El Repository permite que el resto de la aplicación ignore los detalles de persistencia y se centre en la lógica de negocio.

¿Cómo lo usa el ORM?:

El patrón se usa a partir de Session. La Session en su patrón de uso más común comienza en una forma principalmente sin estado. Una vez que se emiten consultas u otros objetos se persisten con ella, la misma solicita un recurso de conexión de un Engine que está asociado con la Session, y luego establece una transacción en esa conexión. Esta transacción permanece en efecto hasta que se instruye a la Session a confirmar o deshacer la transacción. Cuando la transacción termina, el recurso de conexión asociado con el Engine se libera al grupo de conexiones gestionado por el motor. Luego, comienza una nueva transacción con una nueva verificación de conexión. Los objetos ORM mantenidos por una Session están instrumentados de tal manera que cada vez que se modifica un atributo o una colección en el programa Python, se genera un evento de cambio que es registrado por la Session.

Lazy Loading

¿En qué consiste?:

Carga una colección u objeto relacionado según se necesite.

¿Para qué lo usa el ORM?:

Se utiliza cuando accedes a una colección (como una lista de objetos relacionados), SQLAlchemy no carga automáticamente todos los objetos relacionados. En cambio, emite una consulta adicional para cargar los objetos solo cuando los necesitas. Esto es útil para evitar cargar grandes cantidades de datos innecesarios de la base de datos hasta que realmente los requieras.

¿Cómo lo usa el ORM?:

Este comportamiento se puede configurar en el momento de construcción del mapeador utilizando el parámetro relationship.lazy en la función relationship(), así como mediante el uso de opciones de carga ORM

con la construcción Select. La carga de relaciones se divide en tres categorías: carga *perezosa*, carga *ansiosa* y *sin* carga. La carga perezosa se refiere a objetos que se devuelven de una consulta sin que los objetos relacionados se carguen inicialmente. Cuando se accede por primera vez a la colección o referencia dada en un objeto particular, se emite una declaración SELECT adicional para cargar la colección solicitada. Además se pueden usar otras formas de carga como:

- lazy loading - disponible mediante lazy='select' o utilizando la opción lazyload()
- select IN loading - disponible mediante lazy='selectin' o utilizando la opción selectinload()
- joined loading - disponible mediante lazy='joined' o utilizando la opción joinedload()
- raise loading - disponible mediante lazy='raise', o utilizando la opción raiseload()
- subquery loading - disponible mediante lazy='subquery' o utilizando la opción subqueryload()

entre otras.

1.2. Aplicación del ORM

Para nuestro problema hemos decidido utilizar el ORM de SQLAlchemy, el mismo del cual ha tratado nuestro seminario. Entre las razones que nos llevaron a utilizarlo se encuentran:

- Abstracción de la Base de Datos: SQLAlchemy proporciona una capa de abstracción sobre la base de datos relacional, permitiéndonos trabajar con objetos y clases de Python en lugar de escribir consultas SQL directamente. Esto facilita la modelación y manipulación de datos de manera más orientada a objetos.
- Flexibilidad en la Elección de la Base de Datos: SQLAlchemy es compatible con varios sistemas de gestión de bases de datos relacionales como PostgreSQL, MySQL, SQLite y Oracle. En nuestro caso estaremos utilizando como gestor PostgreSQL, pero en un futuro si se decidiera cambiar el gestor de base de datos no tendríamos que modificar las consultas, por lo que se puede utilizar el que mejor se ajuste a las nuevas necesidades y requisitos que puedan surgir en el proceso de desarrollo.
- Modelado de Relaciones Complejas: La empresa de viajes con que estamos trabajando tiene modelos de datos complejos, como son clientes, excursiones, agencias, paquetes, etc. SQLAlchemy permite modelar estas relaciones de manera clara y eficiente, haciendo más fácil trabajar con estructuras.
- Consultas Dinámicas y Avanzadas: SQLAlchemy proporciona un lenguaje expresivo de construcción de consultas que permite realizar consultas dinámicas y avanzadas de manera programática. Esto es beneficioso cuando se necesitan filtrar y recuperar datos específicos de la base de datos.
- Transacciones y Control de Concurrencia: SQLAlchemy maneja de manera eficiente las transacciones y proporciona mecanismos para el control de la concurrencia en entornos multiusuario. Esto es esencial en una aplicación de gestión de viajes donde múltiples usuarios pueden interactuar con la base de datos simultáneamente.
- Escalabilidad y Rendimiento: SQLAlchemy permite optimizar y ajustar el rendimiento de las consultas a medida que la base de datos y la aplicación crecen. Esto es crucial para empresas de viajes que manejan grandes volúmenes de datos y necesitan un rendimiento eficiente.

- Extensibilidad y Personalización: SQLAlchemy es altamente extensible, permitiendo a los desarrolladores personalizar y extender su funcionalidad según sea necesario. Esto es útil cuando la empresa de viajes tiene requisitos específicos que no están cubiertos por las funciones estándar del ORM.