SQLAlchemy

Módulo 1: Introducción a SQLAlchemy

1.1 ¿Qué es SQLAlchemy?

▼ 1.1.1 Definición y Propósito de SQLAlchemy:

SQLAlchemy es una biblioteca de Python que proporciona un conjunto de herramientas flexibles y potentes para trabajar con bases de datos relacionales. Se clasifica como un ORM (Object-Relational Mapping), lo que significa que facilita la interacción con bases de datos relacionales utilizando objetos y clases en lugar de consultas SQL directas.

Propósito:

- **Abstracción de la Base de Datos:** SQLAlchemy proporciona una abstracción eficiente que permite a los desarrolladores interactuar con la base de datos utilizando objetos y clases en lugar de escribir consultas SQL manuales. Esto facilita el manejo de la base de datos y hace que el código sea más legible y mantenible.
- **Portabilidad de Código:** Gracias a su enfoque orientado a objetos, SQLAlchemy permite que el código sea más portable entre diferentes sistemas de gestión de bases de datos (DBMS). Puedes cambiar de una base de datos a otra con mínimos cambios en el código.
- **Mapeo Objeto-Relacional (ORM):** SQLAlchemy ofrece un potente sistema de mapeo objeto-relacional que permite relacionar las estructuras de las tablas de la base de datos con clases de Python. Esto facilita la creación, consulta, actualización y eliminación de registros de la base de datos sin tener que lidiar directamente con SQL.
- **Flexibilidad y Control:** Aunque proporciona un ORM robusto, SQLAlchemy también ofrece un nivel más bajo de acceso a la base de datos para aquellos casos en los que se requiere mayor flexibilidad y control. Esto es especialmente útil en situaciones donde las consultas SQL directas son necesarias.
- **Soporte para Transacciones y Control de Sesiones:** SQLAlchemy facilita la gestión de transacciones y sesiones, lo que es esencial para

mantener la integridad de la base de datos y asegurar que las operaciones se realicen de manera atómica.

- **Integración con Frameworks Web:** SQLAlchemy se integra fácilmente con frameworks web populares como Flask y Django, proporcionando una capa de abstracción para la manipulación de datos en aplicaciones web.

▼ 1.1.2 Comparación con otros ORM (Object-Relational Mapping).

SQLAlchemy no es el único ORM disponible para Python; existen varios otros, cada uno con sus propias características y enfoques. A continuación, se presenta una comparación de SQLAlchemy con algunos de los otros ORM populares:

SQLAlchemy vs. Django ORM:

- **SQLAlchemy:**
  - Ofrece una mayor flexibilidad y control sobre las consultas SQL generadas.
  - Puede ser utilizado de manera independiente sin estar vinculado a un framework específico.
  - Más orientado a la modularidad y se puede integrar fácilmente con diferentes componentes.
- **Django ORM:**
  - Está integrado directamente en el framework Django.
  - Proporciona una sintaxis más simple y rápida para tareas comunes.
  - Es ideal para el desarrollo rápido de aplicaciones web, especialmente cuando se trabaja dentro del ecosistema de Django.

SQLAlchemy vs. Peewee:

- **SQLAlchemy:**
  - Ofrece una abstracción más rica y completa de la base de datos.
  - Es más adecuado para aplicaciones grandes y complejas.
  - Proporciona un nivel de control más detallado sobre las transacciones y las sesiones.
- **Peewee:**

- Es más ligero y fácil de aprender.
- Se enfoca en la simplicidad y la facilidad de uso.
- Puede ser preferido para proyectos más pequeños o cuando la complejidad de SQLAlchemy no es necesaria.

SQLAlchemy vs. Pony ORM:

- **SQLAlchemy:**
  - Es más maduro y ampliamente utilizado en la comunidad de Python.
  - Ofrece una gama más amplia de funcionalidades y opciones de configuración.
  - Proporciona una mayor modularidad, lo que permite utilizar solo las partes necesarias del ORM.
- **Pony ORM:**
  - Tiene una sintaxis más simple y expresiva.
  - Es fácil de aprender y usar para proyectos pequeños a medianos.
  - Ofrece un enfoque más "Pythonic" en la definición de modelos.

SQLAlchemy vs. Tortoise-ORM:

- **SQLAlchemy:**
  - Es más extenso y ofrece una amplia gama de características.
  - Tiene una comunidad más grande y una base de usuarios más establecida.
  - Puede ser más adecuado para proyectos grandes y complejos.
- **Tortoise-ORM:**
  - Está diseñado específicamente para trabajar con bases de datos asincrónicas.
  - Proporciona soporte integrado para bases de datos NoSQL.
  - Es una opción sólida para proyectos que requieren asincronía y escalabilidad.

La elección entre SQLAlchemy y otros ORM depende de los requisitos específicos del proyecto, la preferencia del desarrollador y el ecosistema en el que se esté trabajando. SQLAlchemy destaca por su flexibilidad, control y

extensibilidad, lo que lo hace especialmente adecuado para proyectos más grandes y complejos.

1.2 Instalación y configuración básica

▼ 1.2.1 Instalación de SQLAlchemy.

Para instalar SQLAlchemy, puedes utilizar el administrador de paquetes de Python, `pip`. Asegúrate de tener Python y `pip` instalados en tu sistema antes de seguir estos pasos:

1. **Instalación utilizando pip:**

   Abre tu terminal o línea de comandos y ejecuta el siguiente comando:

   ```
   pip install sqlalchemy
   ```

   Este comando descargará e instalará la última versión estable de SQLAlchemy y sus dependencias.

2. **Verificar la instalación:**

   Puedes verificar que SQLAlchemy se ha instalado correctamente ejecutando el siguiente comando en tu terminal:

   ```
   python -m sqlalchemy --version
   ```

   Esto imprimirá la versión de SQLAlchemy si la instalación fue exitosa.

▼ 1.2.2 Configuración de la conexión a la base de datos.

Para configurar la conexión con la base de datos en SQLAlchemy, necesitarás proporcionar la URL de conexión a tu base de datos. La URL de conexión contiene información como el tipo de base de datos, el nombre de usuario, la contraseña, la dirección del servidor y otros parámetros relevantes. Aquí te muestro cómo hacerlo:

```
from sqlalchemy import create_engine

# Definir la URL de conexión a la base de datos
# Este es un ejemplo para SQLite, pero debes ajustarla s
egún tu base de datos
# SQLite crea la base de datos en un archivo local
```

```
db_url = 'sqlite:///ejemplo.db'

# Crear una instancia de motor (engine)
engine = create_engine(db_url)

# Realizar una conexión a la base de datos
try:
    # Intentar conectarse a la base de datos
    connection = engine.connect()
    print("Conexión exitosa")

    # Aquí puedes realizar operaciones en la base de dat
os

except Exception as e:
    # Si ha habido un problema en la conexión, imprimimo
s el mensaje del error
    print(f"Error de conexión: {e}")

finally:
    # Cerrar la conexión cuando hayas terminado
    if connection:
        connection.close()
```

En este ejemplo, la variable `db_url` contiene la URL de conexión a la base de datos. La URL sigue un formato específico que varía según el tipo de base de datos. En este caso, se usa SQLite para fines ilustrativos, pero para bases de datos como MySQL o PostgreSQL, la URL sería diferente.

Asegúrate de sustituir `example.db` y otros parámetros según la configuración de tu base de datos. Además, ten en cuenta que este código solo establece una conexión y la cierra inmediatamente como ejemplo. En una aplicación real, normalmente mantendrías la conexión abierta mientras la aplicación esté en ejecución y la cerrarías cuando la aplicación termine o cuando ya no necesites acceder a la base de datos.

> Además, para configurar la conexión con Flask, si estás construyendo una aplicación web, puedes usar la

> extensión `SQLAlchemy` de Flask, que simplifica la configuración de la base de datos en el contexto de una aplicación Flask.

Módulo 2: Modelado de Datos con SQLAlchemy

2.1 Definición de Modelos

▼ 2.1.1 Creación de clases de modelo.

En SQLAlchemy, para trabajar con el patrón de mapeo objeto-relacional (ORM), necesitarás definir clases que representen las tablas de tu base de datos. Cada instancia de estas clases será un registro en la base de datos.

A continuación, mostraremos cómo crear clases de modelo utilizando SQLAlchemy y cómo mapearlas a tablas en la base de datos.

Supongamos que tienes una tabla llamada `mi_tabla` con dos columnas: `id` y `nombre`. Aquí tienes un ejemplo básico:

```
from sqlalchemy import create_engine, MetaData
from sqlalchemy import Column,
from sqlalchemy import Integer, String, Datetime

from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

from datetime import datetime

########################################
# Configuración de la base de datos #
########################################

# URL de conexión a la base de datos PostgreSQL
db_url = 'sqlite:///ejemplo.db'

# Crear una instancia de motor (engine)
engine = create_engine(db_url)
```

```
################################
# Creación del modelo de datos #
################################

# Crear una instancia de MetaData
metadata = MetaData()

# Crear la clase de modelo utilizando Declarative Base
Base = declarative_base(metadata = metadata)

# Definir la clase de modelo para la tabla 'mi_tabla'
class MiTabla(Base):
    __tablename__ = 'mi_tabla'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String(30), nullable=False)
        dni = Column(String(9), nullable=False, unique=T
rue)
        date_created = Column(Datetime(), default=dateti
me.utcnow)

# Crear la tabla en la base de datos
metadata.create_all(engine)

# Crear una instancia de sesión
Session = sessionmaker(bind=engine)
session = Session()

# Ejemplo de cómo agregar un registro a la base de datos
nuevo_registro = MiTabla(nombre='Ejemplo')
session.add(nuevo_registro)
session.commit()

# Ejemplo de cómo consultar todos los registros de la ba
se de datos
registros = session.query(MiTabla).all()
```

```
for registro in registros:
    print(f"ID: {registro.id}, Nombre: {registro.nombr
e}")
```

En este ejemplo:

- Se utiliza `declarative_base` para crear una clase base (`Base`) que actuará como base para todas las clases de modelo.
- La clase `MiTabla` hereda de esta clase base y define la estructura de la tabla `mi_tabla`. Las columnas se definen como atributos de clase.
- `__tablename__` especifica el nombre de la tabla en la base de datos.
- `metadata.create_all(engine)` se encarga de crear la tabla en la base de datos.
- Se crea una instancia de sesión para interactuar con la base de datos.

Puedes expandir y personalizar estas clases de modelo según las necesidades de tu aplicación y la estructura de tu base de datos. Además, ten en cuenta que este es solo un ejemplo básico, y en aplicaciones más grandes, es posible que desees organizar y modularizar tu código de manera más eficiente.

▼ 2.1.2 Especificación de tipos de datos.

La jerarquía de tipos

SQLAlchemy proporciona abstracciones para los tipos de datos de bases de datos más comunes, así como varias técnicas para personalizar tipos de datos.

Los tipos de bases de datos se representan mediante clases de Python, las cuales, en última instancia, se extienden desde la clase de tipo base conocida como TypeEngine.

Hay dos categorías generales de tipos de datos, cada una de las cuales se expresa dentro de la jerarquía de tipificación de diferentes maneras. La categoría utilizada por una clase de tipo de datos individual se puede identificar basándose en el uso de dos convenciones de nomenclatura diferentes, que son "CamelCase" y "MAYÚSCULAS".

Los tipos de datos “CamelCase”

Los tipos rudimentarios tienen nombres "CamelCase" como String, Numeric, Integer y DateTime. Todas las subclases inmediatas de TypeEngine son tipos "CamelCase".

Los tipos "CamelCase" son, en el mayor grado posible, independientes de la base de datos, genéricos, lo que significa que todos pueden usarse en cualquier backend de base de datos donde se comportarán de la manera apropiada para ese backend para producir el comportamiento deseado.

Un ejemplo de un tipo de datos sencillo "CamelCase" es **String**. En la mayoría de los backends, el uso de este tipo de datos en una especificación de tabla corresponderá al tipo de base de datos VARCHAR que se utiliza en el backend de destino, entregando valores de cadena hacia y desde la base de datos.

Cuando se utiliza una clase TypeEngine particular en una definición de tabla o en cualquier expresión SQL en general, si no se requieren argumentos, se puede pasar como la clase misma, es decir, sin crear una instancia con (). Si se necesitan argumentos, como el argumento de longitud de 60 en la columna VARCHAR anterior, se puede crear una instancia del tipo.

Otro tipo de datos "CamelCase" que expresa un comportamiento más específico del backend es el tipo de datos booleano. A diferencia de String, que representa un tipo de datos de cadena que tienen todas las bases de datos, no todos los servidores tienen un tipo de datos "booleano" real; algunos utilizan números enteros o valores BIT 0 y 1, algunos tienen constantes literales booleanas verdadero y falso, mientras que otros no. Para este tipo de datos, Boolean puede representar BOOLEAN en un backend como PostgreSQL, BIT en el backend de MySQL y SMALLINT en Oracle. Como los datos se envían y reciben desde la base de datos utilizando este tipo, según el dialecto en uso, puede estar interpretando valores numéricos o booleanos de Python.

Es probable que la aplicación SQLAlchemy típica desee utilizar principalmente tipos “CamelCase” en el caso general, ya que generalmente proporcionarán el mejor comportamiento básico y serán portátiles automáticamente a todos los backends.

La referencia para el conjunto general de tipos de datos “CamelCase” se encuentra en el siguiente enlace:

The Type Hierarchy

current release

https://docs.sqlalchemy.org/en/20/core/type_basics.html#types-generic

Los tipos de datos “UPPERCASE”

A diferencia de los tipos "CamelCase", están los tipos de datos "UPPERCASE". Estos tipos de datos siempre se heredan de un tipo de datos "CamelCase" particular y siempre representan un tipo de datos exacto. Cuando se utiliza un tipo de datos "UPPERCASE", el nombre del tipo siempre se representa exactamente como se indica, sin importar si el backend actual lo admite o no. Por lo tanto, el uso de tipos "UPPERCASE" en una aplicación SQLAlchemy indica que se requieren tipos de datos específicos, lo que implica que la aplicación normalmente, sin tomar pasos adicionales, se limitaría a aquellos servidores que usan el tipo exactamente como se indica.

Ejemplos de tipos UPPERCASE incluyen VARCHAR, NUMERIC, INTEGER y TIMESTAMP, que heredan directamente de los tipos "CamelCase" mencionados anteriormente String, Numeric, Integer y DateTime, respectivamente.

Los tipos de datos "UPPERCASE" que forman parte de sqlalchemy.types son tipos de SQL comunes que normalmente se espera que estén disponibles en al menos dos servidores, si no más.

La referencia para el conjunto general de tipos de datos “UPPERCASE” se encuentra en este enlace:

The Type Hierarchy

current release

https://docs.sqlalchemy.org/en/20/core/type_basics.html#types-sqlstandard

Tipos de datos "UPPERCASE" específicos del backend

La mayoría de las bases de datos también tienen sus propios tipos de datos que son completamente específicos de esas bases de datos o agregan argumentos adicionales que son específicos de esas bases de datos. Para estos tipos de datos, los dialectos específicos de SQLAlchemy proporcionan tipos de datos "UPPERCASE" específicos del backend, para un tipo de SQL que no tiene análogos en otros backends. Ejemplos de tipos de datos en

uppercase específicos del backend incluyen JSONB de PostgreSQL, IMAGE de SQL Server y TINYTEXT de MySQL.

Los backends específicos también pueden incluir tipos de datos "UPPERCASE" que amplían los argumentos disponibles de ese mismo tipo de datos "UPPERCASE" que se encuentra en el módulo sqlalchemy.types. Un ejemplo es al crear un tipo de datos de cadena MySQL, es posible que desee especificar argumentos específicos de MySQL, como charset o national, que están disponibles en la versión MySQL de VARCHAR como parámetros exclusivos de MySQL VARCHAR.charset y VARCHAR.national.

La documentación de API para tipos específicos de backend se encuentra en la documentación específica del dialecto, que se enumera en este enlace:

Dialects
current release
https://docs.sqlalchemy.org/en/20/dialects/index.html

Módulo 3: Consultas con SQLAlchemy

3.1 Consultas Básicas

▼ 3.1.1 Uso de `session.query` para realizar consultas simples.

La función `query` de SQLAlchemy se utiliza para construir y ejecutar consultas en la base de datos.

```
from sqlalchemy import create_engine, Column, Integer, S
tring, ForeignKey, Table
from sqlalchemy.orm import declarative_base, relationshi
p, Session

# Crear el motor y la sesión
engine = create_engine("sqlite:///:memory:", echo=True)
Base = declarative_base()
Base.metadata.create_all(engine)
session = Session(engine)

# Definir modelos
```

```
class Alumno(Base):
    __tablename__ = 'alumnos'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)

    materias = relationship("Materia", secondary=asociac
ion_alumnos_materias, back_populates="alumnos")

class Materia(Base):
    __tablename__ = 'materias'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)

    alumnos = relationship("Alumno", secondary=asociacio
n_alumnos_materias, back_populates="materias")

# Crear registros para estudiantes
estudiante1 = Alumno(nombre='Estudiante1')
estudiante2 = Alumno(nombre='Estudiante2')
estudiante3 = Alumno(nombre='Estudiante3')

# Crear registros para cursos
curso1 = Materia(nombre='Curso1')
curso2 = Materia(nombre='Curso2')
curso3 = Materia(nombre='Curso3')

# Registrar estudiantes y cursos en la base de datos
session.add_all([estudiante1, estudiante2, estudiante3,
curso1, curso2, curso3])
session.commit()

# Uso de session.query para realizar consultas simples
query_result = session.query(Alumno).all()
```

```
# Imprimir resultados
print("Consulta de todos los estudiantes:")
for alumno in query_result:
    print(f"ID: {alumno.id}, Nombre: {alumno.nombre}")
```

En este ejemplo, la consulta `session.query(Alumno).all()` devuelve todos los registros de la tabla de estudiantes (`Alumno`). La función `all()` ejecuta la consulta y obtiene todos los resultados.

Ten en cuenta que este es un ejemplo simple sin filtrado. La variable `query_result` contendrá una lista de objetos `Alumno` que representan todos los estudiantes en la base de datos, y luego se imprime la información de estos estudiantes.

En el siguiente punto, agregaremos filtrado a las consultas para obtener resultados más específicos.

▼ 3.1.2 Filtrado de resultados.

En SQLAlchemy, puedes utilizar métodos como `filter`, `filter_by` y `filter` aplicados a la consulta para agregar condiciones de filtrado.

```
# Uso de session.query con filtrado
# Filtrar estudiantes cuyo nombre sea 'Estudiante1'
filtered_result = session.query(Alumno).filter(Alumno.no
mbre == 'Estudiante1').all()

# Imprimir resultados filtrados
print("\nConsulta filtrada de estudiantes:")
for alumno in filtered_result:
    print(f"ID: {alumno.id}, Nombre: {alumno.nombre}")
```

En este ejemplo, la consulta `session.query(Alumno).filter(Alumno.nombre == 'Estudiante1').all()` filtra los estudiantes cuyo nombre es 'Estudiante1'. La función `filter` se utiliza para aplicar condiciones de filtrado.

Puedes experimentar con diferentes condiciones de filtrado y combinaciones de filtros para obtener resultados más específicos según tus necesidades. Aquí, solo he mostrado un ejemplo simple.

3.2 Consultas Avanzadas

▼ 3.2.1 Uso de operadores de comparación.

En SQLAlchemy, puedes utilizar operadores de comparación para realizar consultas avanzadas que involucren condiciones más complejas. A continuación, te mostraré cómo puedes utilizar algunos operadores de comparación comunes.

```
from sqlalchemy import create_engine, Column, Integer, S
tring, ForeignKey, Table
from sqlalchemy.orm import declarative_base, relationshi
p, Session

# Crear el motor y la sesión
engine = create_engine("sqlite:///:memory:", echo=True)
Base = declarative_base()
Base.metadata.create_all(engine)
session = Session(engine)

# Definir modelos
class Producto(Base):
    __tablename__ = 'productos'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)
    precio = Column(Integer)
    stock = Column(Integer)

# Crear algunos registros de productos
producto1 = Producto(nombre='Laptop', precio=1000, stock
=5)
producto2 = Producto(nombre='Teléfono', precio=500, stoc
k=10)
producto3 = Producto(nombre='Tableta', precio=300, stock
=8)

session.add_all([producto1, producto2, producto3])
session.commit()

# Uso de operadores de comparación en consultas avanzada
```

```
s
# Ejemplo 1: Productos con un precio mayor a 500
resultados_precio_mayor_500 = session.query(Producto).fi
lter(Producto.precio > 500).all()
resultados_precio_mayor_500 = session.query(Producto).fi
lter_by(precio > 500).all()

print("\\nProductos con precio mayor a 500:")
for producto in resultados_precio_mayor_500:
    print(f"ID: {producto.id}, Nombre: {producto.nombr
e}, Precio: {producto.precio}")

# Ejemplo 2: Productos con stock menor o igual a 5
resultados_stock_menor_igual_5 = session.query(Product
o).filter(Producto.stock <= 5).all()

print("\\nProductos con stock menor o igual a 5:")
for producto in resultados_stock_menor_igual_5:
    print(f"ID: {producto.id}, Nombre: {producto.nombr
e}, Stock: {producto.stock}")
```

En estos ejemplos, se utilizan operadores de comparación como `>`, `<=` para filtrar resultados basados en condiciones específicas. Puedes experimentar con otros operadores de comparación según tus necesidades.

▼ 3.2.2 Ordenación de resultados.

En SQLAlchemy, puedes utilizar el método `order_by()` para especificar el orden en el que deseas que se devuelvan los resultados de una consulta. Aquí te muestro cómo puedes ordenar los resultados de una consulta utilizando este método:

```
from sqlalchemy import create_engine, Column, Integer, S
tring, ForeignKey, Table
from sqlalchemy.orm import declarative_base, relationshi
p, Session

# Crear el motor y la sesión
engine = create_engine("sqlite:///:memory:", echo=True)
Base = declarative_base()
```

```
Base.metadata.create_all(engine)
session = Session(engine)

# Definir modelos
class Producto(Base):
    __tablename__ = 'productos'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)
    precio = Column(Integer)
    stock = Column(Integer)

# Crear algunos registros de productos
producto1 = Producto(nombre='Laptop', precio=1000, stock
=5)
producto2 = Producto(nombre='Teléfono', precio=500, stoc
k=10)
producto3 = Producto(nombre='Tableta', precio=300, stock
=8)

session.add_all([producto1, producto2, producto3])
session.commit()

# Ordenar resultados por precio de forma ascendente
resultados_ordenados_precio_asc = session.query(Product
o).order_by(Producto.precio).all()

print("\\nProductos ordenados por precio de forma ascend
ente:")
for producto in resultados_ordenados_precio_asc:
    print(f"ID: {producto.id}, Nombre: {producto.nombr
e}, Precio: {producto.precio}")

# Ordenar resultados por stock de forma descendente
resultados_ordenados_stock_desc = session.query(Product
o).filter(Producto.precio > 500).all().order_by(Product
o.stock.desc())
```

```
print("\\nProductos ordenados por stock de forma descend
ente:")
for producto in resultados_ordenados_stock_desc:
    print(f"ID: {producto.id}, Nombre: {producto.nombr
e}, Stock: {producto.stock}")
```

En estos ejemplos, se utiliza `order_by()` para ordenar los resultados de la consulta. En el primer caso, se ordenan los productos por precio de forma ascendente, y en el segundo caso, se ordenan por stock de forma descendente.

Puedes modificar la consulta según tus necesidades y utilizar `asc()` o `desc()` para especificar la dirección de la ordenación. Además, puedes ordenar por múltiples columnas al pasar varias columnas como argumentos a `order_by()`.

Recuerda que estos son solo ejemplos básicos, y puedes adaptarlos según los requisitos específicos de tu aplicación.

▼ 3.2.3 Consultas con funciones de agregación.

Las funciones de agregación se pueden utilizar para realizar operaciones sobre conjuntos de datos, como sumas, promedios, contar registros, entre otras.

Aquí mostramos cómo puedes realizar consultas con funciones de agregación:

```
from sqlalchemy import create_engine, Column, Integer, S
tring, ForeignKey, func
from sqlalchemy.orm import declarative_base, relationshi
p, Session

# Crear el motor y la sesión
engine = create_engine("sqlite:///:memory:", echo=True)
Base = declarative_base()
Base.metadata.create_all(engine)
session = Session(engine)

# Definir modelos
class Venta(Base):
```

```
    __tablename__ = 'ventas'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    producto = Column(String)
    cantidad = Column(Integer)
    precio_unitario = Column(Integer)

# Crear algunos registros de ventas
venta1 = Venta(producto='Laptop', cantidad=2, precio_uni
tario=1000)
venta2 = Venta(producto='Teléfono', cantidad=5, precio_u
nitario=500)
venta3 = Venta(producto='Tableta', cantidad=3, precio_un
itario=300)

session.add_all([venta1, venta2, venta3])
session.commit()

# Consultas con funciones de agregación
# Obtener la cantidad total de productos vendidos
total_cantidad_vendida = session.query(func.sum(Venta.ca
ntidad)).scalar()
print("\\nTotal de cantidad de productos vendidos:", tot
al_cantidad_vendida)

# Obtener el precio promedio de los productos vendidos
precio_promedio = session.query(func.avg(Venta.precio_un
itario)).scalar()
print("Precio promedio de productos vendidos:", precio_p
romedio)

# Obtener la cantidad de ventas realizadas
total_ventas = session.query(func.count(Venta.id)).scala
r()
print("Total de ventas realizadas:", total_ventas)
```

En este ejemplo, se utilizan funciones de agregación como `func.sum()`, `func.avg()`, y `func.count()` en las consultas. La función `scalar()` se utiliza para obtener el resultado de la función de agregación.

Módulo 4: Manipulación de Datos con SQLAlchemy

▼ 4.1 Inserción de Datos

Uso de `session.add()` para insertar nuevos registros.

Ahora, procederemos a mostrar cómo usar `session.add()` para insertar nuevos registros en la base de datos. `session.add()` se utiliza para agregar nuevos objetos a la sesión, y estos objetos se insertarán en la base de datos cuando se realice un commit en la sesión.

```
# Crear un nuevo estudiante y agregarlo a la sesión
nuevo_estudiante = Alumno(nombre='NuevoEstudiante')
session.add(nuevo_estudiante)

# Realizar commit para insertar el nuevo estudiante en
la base de datos
session.commit()

# Consultar todos los estudiantes después de la inserc
ión
todos_los_estudiantes = session.query(Alumno).all()

# Imprimir resultados
print("\\nTodos los estudiantes después de la inserció
n:")
for alumno in todos_los_estudiantes:
    print(f"ID: {alumno.id}, Nombre: {alumno.nombre}")
```

En este ejemplo, creamos un nuevo objeto `Alumno` con el nombre 'NuevoEstudiante' y lo agregamos a la sesión utilizando `session.add()`. Posteriormente, realizamos un commit para insertar el nuevo estudiante en la base de datos. Después, realizamos una consulta para obtener

todos los estudiantes y mostramos los resultados, incluido el nuevo estudiante.

Este es un proceso básico para agregar nuevos registros a la base de datos utilizando SQLAlchemy. Puedes adaptar este enfoque para trabajar con otros modelos y agregar datos según sea necesario en tu aplicación.

▼ 4.2 Actualización de Datos

Uso de `session.commit()` para actualizar registros existentes.

`session.commit()` se utiliza para confirmar (guardar) todas las operaciones pendientes en la sesión, incluyendo inserciones, actualizaciones y eliminaciones. Para actualizar registros existentes, primero debes cargar el registro que deseas modificar, realizar los cambios y luego confirmarlos mediante `session.commit()`.

Aquí hay un ejemplo de cómo puedes usar `session.commit()` para actualizar el nombre de un estudiante existente:

```
# Consultar un estudiante específico por nombre
estudiante_a_actualizar = session.query(Alumno).filter
(Alumno.nombre == 'Estudiante1').first()

# Imprimir información antes de la actualización
print("\\nInformación del estudiante antes de la actua
lización:")
print(f"ID: {estudiante_a_actualizar.id}, Nombre: {est
udiante_a_actualizar.nombre}")

# Actualizar el nombre del estudiante
estudiante_a_actualizar.nombre = 'NuevoNombreEstudiant
e1'

# Realizar commit para aplicar la actualización
session.commit()

# Consultar el estudiante después de la actualización
estudiante_actualizado = session.query(Alumno).filter
(Alumno.id == estudiante_a_actualizar.id).first()

# Imprimir información después de la actualización
```

```
print("\\nInformación del estudiante después de la act
ualización:")
print(f"ID: {estudiante_actualizado.id}, Nombre: {estu
diante_actualizado.nombre}")
```

En este ejemplo:

1. Consultamos un estudiante específico utilizando `session.query()` y `filter`.
2. Imprimimos la información del estudiante antes de la actualización.
3. Modificamos el nombre del estudiante.
4. Realizamos `session.commit()` para aplicar la actualización en la base de datos.
5. Consultamos el estudiante después de la actualización y mostramos la información actualizada.

Recuerda que `session.commit()` debe usarse con precaución, ya que confirma todos los cambios pendientes en la sesión. Asegúrate de aplicar actualizaciones solo cuando sea necesario y después de haber verificado que los cambios son correctos.

▼ 4.3 Eliminación de Datos

El método `session.delete()` se utiliza para eliminar registros de la base de datos. Aquí hay un ejemplo de cómo puedes usar para eliminar un estudiante específico:

Uso de `session.delete()` para eliminar registros.

```
pythonCopy code
# Crear un nuevo estudiante y agregarlo a la sesión
nuevo_estudiante = Alumno(nombre='NuevoEstudiante')
session.add(nuevo_estudiante)

# Realizar commit para insertar el nuevo estudiante en
la base de datos
session.commit()

# Consultar todos los estudiantes antes de la eliminac
```

```
ión
todos_los_estudiantes_antes = session.query(Alumno).al
l()

# Imprimir resultados antes de la eliminación
print("Todos los estudiantes antes de la eliminació
n:")
for alumno in todos_los_estudiantes_antes:
    print(f"ID: {alumno.id}, Nombre: {alumno.nombre}")

# Eliminar el estudiante recién creado
session.delete(nuevo_estudiante)
session.commit()

# Consultar todos los estudiantes después de la elimin
ación
todos_los_estudiantes_despues = session.query(Alumno).
all()

# Imprimir resultados después de la eliminación
print("\nTodos los estudiantes después de la eliminaci
ón:")
for alumno in todos_los_estudiantes_despues:
    print(f"ID: {alumno.id}, Nombre: {alumno.nombre}")
```

En este ejemplo, creamos un nuevo estudiante, lo agregamos a la sesión y realizamos un commit para insertarlo en la base de datos. Luego, consultamos todos los estudiantes antes de la eliminación y mostramos los resultados. Posteriormente, utilizamos `session.delete()` para eliminar el estudiante recién creado y volvemos a realizar un commit. Finalmente, consultamos todos los estudiantes después de la eliminación y mostramos los resultados actualizados.

Módulo 5: Transacciones y Control de Sesiones

▼ 5.1 Transacciones

Uso de `session.begin()` y `session.commit()` para gestionar transacciones.

Las transacciones en SQLAlchemy se gestionan utilizando `session.begin()` y `session.commit()`. Aquí tienes un ejemplo básico de cómo puedes utilizar estas funciones para gestionar transacciones:

```
from sqlalchemy import create_engine, Column, Integer,
String
from sqlalchemy.orm import declarative_base, Session

# Crear el motor y la sesión
engine = create_engine("sqlite:///:memory:", echo=Tru
e)
Base = declarative_base()
Base.metadata.create_all(engine)
session = Session(engine)

# Definir un modelo simple
class Producto(Base):
    __tablename__ = 'productos'

    id = Column(Integer, primary_key=True, autoincreme
nt=True)
    nombre = Column(String)
    precio = Column(Integer)
    stock = Column(Integer)

# Función para imprimir todos los productos
def imprimir_productos():
    print("\\nTodos los productos en la base de dato
s:")
    productos = session.query(Producto).all()
    for producto in productos:
        print(f"ID: {producto.id}, Nombre: {producto.n
ombre}, Precio: {producto.precio}, Stock: {producto.st
ock}")

# Crear algunos registros de productos
producto1 = Producto(nombre='Laptop', precio=1000, sto
```

```
ck=5)
producto2 = Producto(nombre='Teléfono', precio=500, st
ock=10)

# Iniciar una transacción
with session.begin():
    # Agregar productos a la base de datos
    session.add_all([producto1, producto2])

# Imprimir productos después de la transacción
imprimir_productos()

# Actualizar el stock de un producto
producto_a_actualizar = session.query(Producto).filter
(Producto.nombre == 'Laptop').first()
producto_a_actualizar.stock = 8

# Iniciar una nueva transacción
with session.begin():
    # Actualizar el producto en la base de datos
    session.add(producto_a_actualizar)

# Imprimir productos después de la segunda transacción
imprimir_productos()
```

En este ejemplo:

1. Se utiliza `with session.begin()` para iniciar una transacción.
2. Se agregan algunos productos a la base de datos dentro de la transacción.
3. Después de la transacción, se imprime la lista de productos.
4. Se actualiza el stock de un producto.
5. Se inicia una nueva transacción para aplicar la actualización.
6. Después de la segunda transacción, se imprime la lista de productos actualizada.

La utilización de `session.begin()` y `with session.begin()` asegura que las transacciones se gestionen correctamente. La transacción se confirma

automáticamente si no hay errores dentro del bloque `with`. Si ocurre una excepción, la transacción se revierte automáticamente. Esto ayuda a mantener la consistencia de la base de datos.

▼ 5.2 Control de Sesiones

Manejo de sesiones y contextos.

En SQLAlchemy, el manejo de sesiones y contextos es crucial para garantizar la consistencia y la correcta administración de la base de datos. La gestión de sesiones suele ir de la mano con el uso de contextos para garantizar que las operaciones de la base de datos se realicen de manera segura. Aquí hay un ejemplo básico que ilustra cómo manejar sesiones y contextos en SQLAlchemy:

```
from sqlalchemy import create_engine, Column, Integer,
String
from sqlalchemy.orm import declarative_base, Session,
sessionmaker

# Crear el motor y la sesión
engine = create_engine("sqlite:///:memory:", echo=Tru
e)
Base = declarative_base()
Base.metadata.create_all(engine)

# Definir un modelo simple
class Producto(Base):
    __tablename__ = 'productos'

    id = Column(Integer, primary_key=True, autoincreme
nt=True)
    nombre = Column(String)
    precio = Column(Integer)
    stock = Column(Integer)

# Crear una fábrica de sesiones utilizando sessionmake
r
Session = sessionmaker(bind=engine)
```

```
# Ejemplo de uso de sesiones y contextos
def ejemplo_uso_sesiones():
    # Abrir una nueva sesión
    with Session() as session:
        # Realizar operaciones en la base de datos
        producto = Producto(nombre='Laptop', precio=10
00, stock=5)
        session.add(producto)

        # Imprimir productos después de agregar uno
        productos = session.query(Producto).all()
        print("\\nProductos después de agregar uno:")
        for p in productos:
            print(f"ID: {p.id}, Nombre: {p.nombre}, Pr
ecio: {p.precio}, Stock: {p.stock}")

    # Intentar imprimir productos fuera del contexto d
e la sesión
    try:
        productos_fuera_del_contexto = session.query(P
roducto).all()
    except Exception as e:
        print(f"Error al intentar imprimir productos f
uera del contexto de sesión: {e}")

# Llamar a la función de ejemplo
ejemplo_uso_sesiones()
```

En este ejemplo:

1. Se utiliza `sessionmaker` para crear una fábrica de sesiones (`Session`) que está vinculada al motor de la base de datos.
2. Se abre una nueva sesión utilizando el contexto `with Session() as session:`. Esto garantiza que la sesión se cierre adecuadamente al final del bloque `with`.
3. Dentro del bloque `with`, se realizan operaciones en la base de datos, como agregar un nuevo producto.

4. Después de salir del bloque `with`, se intenta realizar una consulta fuera del contexto de la sesión para demostrar cómo las operaciones deben realizarse dentro del contexto para garantizar la correcta administración de la sesión.

Es importante utilizar contextos (`with`) para garantizar que las sesiones se cierren correctamente y que las transacciones se gestionen adecuadamente. Además, la gestión de contextos permite un código más limpio y seguro en la interacción con la base de datos.

Módulo 6: Relaciones entre Modelos

▼ 6.1 Uso de claves primarias y foráneas.

Al utilizar SQLAlchemy, es importante entender cómo trabajar con claves primarias y foráneas para establecer relaciones entre tablas. A continuación, se describen los conceptos clave relacionados con claves primarias y foráneas:

1. Definición de Claves Primarias:

La clave primaria de una tabla es una columna (o un conjunto de columnas) que identifica de manera única cada fila en la tabla. En SQLAlchemy, puedes definir una clave primaria utilizando el atributo `primary_key=True` en la columna correspondiente.

Ejemplo:

```
class Alumno(Base):
    __tablename__ = 'alumnos'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)
```

En este ejemplo, `id` es la clave primaria de la tabla `alumnos`.

2. Definición de Claves Foráneas:

Una clave foránea es una columna que establece una relación entre dos tablas. La clave foránea en una tabla hace referencia a la clave primaria de

otra tabla. En SQLAlchemy, puedes definir una clave foránea utilizando `ForeignKey` y especificando la tabla y la columna a la que hace referencia.

Ejemplo:

```
class Inscripcion(Base):
    __tablename__ = 'inscripciones'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    alumno_id = Column(Integer, ForeignKey('alumnos.i
d'))
    materia_id = Column(Integer, ForeignKey('materias.i
d'))

    alumno = relationship("Alumno", back_populates="mate
rias")
    materia = relationship("Materia", back_populates="al
umnos")
```

En este ejemplo, `alumno_id` y `materia_id` son claves foráneas que hacen referencia a las claves primarias de las tablas `alumnos` y `materias`, respectivamente.

3. Uso de Claves Primarias y Foráneas en Relaciones Muchos a Muchos:

Cuando trabajas con relaciones muchos a muchos, utilizas una tabla de asociación que contiene claves foráneas que hacen referencia a las tablas relacionadas. Las claves foráneas en la tabla de asociación establecen vínculos entre las tablas.

Ejemplo:

```
asociacion_alumnos_materias = Table(
    'asociacion_alumnos_materias',
    Base.metadata,
    Column('alumno_id', Integer, ForeignKey('alumnos.i
d')),
    Column('materia_id', Integer, ForeignKey('materias.i
```

```
d'))
)
```

En este ejemplo, `alumno_id` y `materia_id` son claves foráneas en la tabla de asociación `asociacion_alumnos_materias`.

4. Uso de `relationship` para Mapear Relaciones:

La función `relationship` en SQLAlchemy se utiliza para mapear relaciones entre tablas. Se utiliza en conjunto con las claves primarias y foráneas para establecer las relaciones y permitir el acceso a través de objetos Python.

Ejemplo:

```
class Alumno(Base):
    # ... (definición de la clase)

    materias = relationship("Materia", secondary=asociac
ion_alumnos_materias, back_populates="alumnos")

class Materia(Base):
    # ... (definición de la clase)

    alumnos = relationship("Alumno", secondary=asociacio
n_alumnos_materias, back_populates="materias")
```

En este ejemplo, `relationship` se utiliza para mapear la relación muchos a muchos entre `Alumno` y `Materia` a través de la tabla de asociación `asociacion_alumnos_materias`.

▼ 6.2 Definición de relaciones uno a uno, uno a muchos y muchos a muchos.

En SQLAlchemy, puedes definir relaciones entre las tablas utilizando el concepto de relaciones en el modelo objeto-relacional (ORM).

En estos ejemplos, se utilizan las funciones `relationship` y `ForeignKey` de SQLAlchemy para definir las relaciones.

A continuación, mostramos cómo definir relaciones uno a uno, uno a muchos y muchos a muchos.

1. Relación Uno a Uno:

En una relación uno a uno, cada registro en la tabla A se relaciona con exactamente un registro en la tabla B, y viceversa. Aquí hay un ejemplo:

```
from sqlalchemy import Column, Integer, String, ForeignK
ey
from sqlalchemy.orm import relationship

class Persona(Base):
    __tablename__ = 'personas'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)

    # Definir relación uno a uno con la tabla Direccione
s
    direccion = relationship("Direccion", uselist=False,
back_populates="persona")

class Direccion(Base):
    __tablename__ = 'direcciones'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    calle = Column(String)
    ciudad = Column(String)

    # Definir relación uno a uno con la tabla Personas
    persona_id = Column(Integer, ForeignKey('personas.i
d'))
    persona = relationship("Persona", back_populates="di
reccion")
```

2. Relación Uno a Muchos:

En una relación uno a muchos, un registro en la tabla A puede estar relacionado con varios registros en la tabla B, pero cada registro en la tabla B está relacionado con solo un registro en la tabla A.

Aquí hay un ejemplo enel que la tabla `Inscripcion` se usa como pivote para establecer una relación entre los cursos y los estudiantes de manera que cada inscripción corresponde con un sólo estudiante y un sólo curso aunque un curso o un estudiante puedan tener muchas inscripciones

```
from sqlalchemy import Column, Integer, String, ForeignK
ey
from sqlalchemy.orm import relationship

class Curso(Base):
    __tablename__ = 'cursos'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)

class Estudiante(Base):
    __tablename__ = 'estudiantes'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)

    # Definir relación uno a muchos con la tabla Cursos
    cursos = relationship("Curso", back_populates="estud
iante")

class Inscripcion(Base):
    __tablename__ = 'inscripciones'

    id = Column(Integer, primary_key=True, autoincrement
=True)

    # Definir relación uno a muchos con la tabla Estudia
ntes
    estudiante_id = Column(Integer, ForeignKey('estudian
tes.id'))
    estudiante = relationship("Estudiante", back_populat
```

```
es="inscripciones")

    # Definir relación uno a muchos con la tabla Cursos
    curso_id = Column(Integer, ForeignKey('cursos.id'))
    curso = relationship("Curso", back_populates="inscri
pciones")
```

3. Relación Muchos a Muchos:

En una relación muchos a muchos, un registro en la tabla A puede estar relacionado con varios registros en la tabla B, y viceversa.

En este ejemplo:

- La clase `Alumno` representa a los estudiantes.
- La clase `Materia` representa a las asignaturas o cursos.
- La tabla de asociación se llama `asociacion_alumnos_materias`.
- La relación muchos a muchos se establece mediante las propiedades `materias` en la clase `Alumno` y `alumnos` en la clase `Materia`.

```
from sqlalchemy import Column, Integer, String, Table, F
oreignKey
from sqlalchemy.orm import relationship

# Crear una tabla de asociación para la relación muchos
a muchos
asociacion_alumnos_materias = Table(
    'asociacion_alumnos_materias',
    Base.metadata,
    Column('alumno_id', Integer, ForeignKey('alumnos.i
d')),
    Column('materia_id', Integer, ForeignKey('materias.i
d'))
)

class Alumno(Base):
    __tablename__ = 'alumnos'

    id = Column(Integer, primary_key=True, autoincrement
```

```
=True)
    nombre = Column(String)

    # Definir relación muchos a muchos con la tabla Mate
rias
    materias = relationship("Materia", secondary=asociac
ion_alumnos_materias, back_populates="alumnos")

class Materia(Base):
    __tablename__ = 'materias'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)

    # Definir relación muchos a muchos con la tabla Alum
nos
    alumnos = relationship("Alumno", secondary=asociacio
n_alumnos_materias, back_populates="materias")
```

Módulo 7: Mapeo Avanzado y Extensiones

▼ 7.1 Uso de herencia en la definición de modelos.

En SQLAlchemy, puedes utilizar el mapeo de herencia para modelar jerarquías de clases en tu base de datos. Hay varias estrategias para implementar la herencia en SQLAlchemy, y una de ellas es la herencia de tabla única (single table inheritance). En este enfoque, todas las clases de la jerarquía comparten la misma tabla en la base de datos.

A continuación, te presento un ejemplo básico de cómo puedes utilizar la herencia de tabla única en SQLAlchemy:

```
from sqlalchemy import create_engine, Column, Integer, S
tring, ForeignKey
from sqlalchemy.orm import declarative_base, Session, re
lationship
```

```
# Crear el motor y la sesión
engine = create_engine("sqlite:///:memory:", echo=True)
Base = declarative_base()
Base.metadata.create_all(engine)
session = Session(engine)

# Definir modelos con herencia de tabla única
class Animal(Base):
    __tablename__ = 'animales'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)
    tipo = Column(String)

    __mapper_args__ = {
        'polymorphic_identity': 'animal',
        'polymorphic_on': tipo
    }

class Perro(Animal):
    __tablename__ = 'perros'

    id = Column(Integer, ForeignKey('animales.id'), prim
ary_key=True)
    raza = Column(String)

    __mapper_args__ = {
        'polymorphic_identity': 'perro'
    }

class Gato(Animal):
    __tablename__ = 'gatos'

    id = Column(Integer, ForeignKey('animales.id'), prim
ary_key=True)
    color = Column(String)
```

```
    __mapper_args__ = {
        'polymorphic_identity': 'gato'
    }

# Crear algunos registros de animales
animal1 = Animal(nombre='Animal1')
perro1 = Perro(nombre='Perro1', raza='Labrador')
gato1 = Gato(nombre='Gato1', color='Negro')

session.add_all([animal1, perro1, gato1])
session.commit()

# Consultar todos los animales
todos_los_animales = session.query(Animal).all()
print("\\nTodos los animales en la base de datos:")
for animal in todos_los_animales:
    print(f"ID: {animal.id}, Nombre: {animal.nombre}, Ti
po: {animal.tipo}")
    if isinstance(animal, Perro):
        print(f"   Raza: {animal.raza}")
    elif isinstance(animal, Gato):
        print(f"   Color: {animal.color}")
```

En este ejemplo:

- Se define la clase base `Animal` con atributos comunes y un identificador de tipo (`tipo`) que se utiliza para diferenciar entre perros y gatos.
- Se definen las clases hijas `Perro` y `Gato` que heredan de la clase `Animal` y agregan atributos específicos.
- Se utilizan las opciones `polymorphic_identity` y `polymorphic_on` en el mapeo para indicar la identidad polimórfica y la columna sobre la cual se basa la discriminación de tipo.
- Se crean registros de animales que pueden ser de tipo `Animal`, `Perro`, o `Gato`.
- Se consultan todos los animales en la base de datos y se imprime la información.

Este es solo un ejemplo de la herencia de tabla única en SQLAlchemy. Dependiendo de tus necesidades específicas, podrías optar por otras estrategias de mapeo de herencia, como la herencia de tabla concreta o la herencia de clase abstracta.

▼ 7.2 Uso de extensiones y eventos para personalizar el comportamiento de SQLAlchemy.

En SQLAlchemy, las extensiones y eventos te permiten personalizar y extender el comportamiento de la biblioteca de ORM. Puedes utilizar extensiones para agregar funcionalidades adicionales a SQLAlchemy, y los eventos te permiten ejecutar código en respuesta a ciertos eventos específicos.

Aquí te proporciono un ejemplo básico de cómo puedes usar una extensión y un evento en SQLAlchemy:

```
from sqlalchemy import create_engine, Column, Integer, S
tring, event
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import Session

# Crear el motor y la sesión
engine = create_engine("sqlite:///:memory:", echo=True)
Base = declarative_base()
Base.metadata.create_all(engine)
session = Session(engine)

# Definir un modelo simple
class Producto(Base):
    __tablename__ = 'productos'

    id = Column(Integer, primary_key=True, autoincrement
=True)
    nombre = Column(String)
    precio = Column(Integer)
    stock = Column(Integer)

# Extensión: Uso de una columna calculada para el valor
total del producto
```

```
class ProductoConTotal(Producto):
    total = Column(Integer)

    def calcular_total(self):
        self.total = self.precio * self.stock

# Evento: Antes de insertar un producto en la base de da
tos, calcular el total
@event.listens_for(ProductoConTotal, 'before_insert')
def before_insert_listener(mapper, connection, target):
    target.calcular_total()

# Crear algunos registros de productos utilizando la cla
se extendida
producto1 = ProductoConTotal(nombre='Laptop', precio=100
0, stock=5)
producto2 = ProductoConTotal(nombre='Teléfono', precio=5
00, stock=10)

session.add_all([producto1, producto2])
session.commit()

# Consultar y imprimir productos con sus totales
productos_con_totales = session.query(ProductoConTotal).
all()
print("\\nProductos con totales después de la inserció
n:")
for producto in productos_con_totales:
    print(f"ID: {producto.id}, Nombre: {producto.nombr
e}, Precio: {producto.precio}, Stock: {producto.stock},
Total: {producto.total}")
```

En este ejemplo:

1. Se define una extensión mediante la creación de una clase `ProductoConTotal` que hereda de `Producto` y agrega una columna calculada `total`.
2. Se define un evento utilizando `@event.listens_for` que escucha el evento `'before_insert'`. Este evento se dispara antes de insertar un nuevo objeto

en la base de datos.

3. Se implementa una función `before_insert_listener` que calcula el total del producto antes de la inserción.

Módulo 8: Rendimiento y Optimización

▼ 8.1 Uso de estrategias de carga para optimizar consultas.

Las estrategias de carga en SQLAlchemy te permiten optimizar las consultas para minimizar la cantidad de consultas a la base de datos y mejorar el rendimiento de tu aplicación. A continuación, te presento algunas de las estrategias de carga más comunes:

1. **selectinload():** Esta estrategia de carga se utiliza para cargar una relación específica junto con la consulta principal en una sola consulta. Esto es útil cuando tienes un conjunto de objetos relacionados que deseas cargar en una única consulta.

```
from sqlalchemy.orm import selectinload

# Ejemplo de uso de selectinload
usuarios = session.query(Usuario).options(selectinloa
d(Usuario.direcciones)).all()
```

2. **joinedload():** Esta estrategia de carga realiza una unión (JOIN) en la base de datos para recuperar la relación junto con la consulta principal. Puede ser útil cuando tienes relaciones que no están cubiertas por selectinload() y deseas cargar los datos relacionados en una sola consulta.

```
from sqlalchemy.orm import joinedload

# Ejemplo de uso de joinedload
usuarios = session.query(Usuario).options(joinedload
(Usuario.direcciones)).all()
```

3. **subqueryload():** Esta estrategia de carga utiliza una subconsulta para cargar la relación. Es útil cuando la relación no se puede cargar eficientemente con selectinload() o joinedload().

```
from sqlalchemy.orm import subqueryload

# Ejemplo de uso de subqueryload
usuarios = session.query(Usuario).options(subqueryloa
d(Usuario.direcciones)).all()
```

4. **lazy loading:** Puedes especificar la carga diferida (lazy loading) para una relación, lo que significa que los datos de la relación se cargarán solo cuando se acceda a ellos por primera vez. Esto puede ser útil si deseas cargar datos de manera diferida para mejorar el rendimiento de consultas iniciales.

```
from sqlalchemy.orm import relationship

class Usuario(Base):
    # ...
    direcciones = relationship('Direccion', lazy='sel
ect')
```

Estas estrategias de carga te brindan flexibilidad para ajustar el comportamiento de carga según tus necesidades específicas. La elección de la estrategia adecuada depende del contexto de tu aplicación, la cantidad de datos y la eficiencia de las consultas que deseas lograr. Experimenta con estas estrategias y evalúa su impacto en el rendimiento de tu aplicación.

▼ 8.2 Uso de caché y memoria para mejorar el rendimiento.

El uso de caché y técnicas de gestión de memoria puede ser crucial para mejorar el rendimiento de tu aplicación. SQLAlchemy ofrece algunas características que pueden ayudarte en este aspecto. A continuación, se presentan algunas estrategias que puedes considerar:

1. **Cacheando consultas:**

- **Result Caching:** SQLAlchemy ofrece un sistema de caché de resultados que puede ayudar a evitar consultas redundantes a la base de datos. Puedes habilitar el caché de resultados utilizando la opción `cache_from_source` en la consulta.

```
from sqlalchemy.orm import Query

# Ejemplo de caché de resultados
query = session.query(Usuario).options(Query.cache
_from_source("orm")).filter_by(id=1)
user = query.first()
```

- **Query Memoization:** Puedes utilizar técnicas de memoización para almacenar en caché el resultado de consultas específicas.

```
from functools import lru_cache

@lru_cache(maxsize=None)
def obtener_usuario_por_id(id_usuario):
    return session.query(Usuario).filter_by(id=id_
usuario).first()
```

2. **Caché de objetos:**
   - **Identity Map:** SQLAlchemy utiliza un "Identity Map" internamente para garantizar que cada objeto en una sesión se cargue solo una vez. Este enfoque ayuda a evitar problemas de inconsistencia al trabajar con múltiples instancias del mismo objeto.
3. **Optimizaciones de memoria:**
   - **Session.expunge_all():** Si tienes una gran cantidad de objetos cargados en una sesión y ya no los necesitas, puedes utilizar `session.expunge_all()` para eliminar todos los objetos de la sesión. Esto puede ayudar a liberar memoria.

```
session.expunge_all()
```

   - **Lazy Loading:** Utiliza cargas diferidas (lazy loading) para cargar solo los datos que realmente necesitas en un momento dado. Esto puede

evitar cargar grandes conjuntos de datos en memoria de manera innecesaria.

4. **Caché de consulta en segundo plano:**
    - Algunas bibliotecas de caché externas, como Redis o Memcached, pueden utilizarse para almacenar en caché resultados de consultas y datos frecuentemente utilizados. SQLAlchemy no proporciona integración directa con estas bibliotecas, pero puedes implementar lógica personalizada para almacenar en caché y recuperar resultados según sea necesario.

Es importante ajustar estas estrategias según las necesidades específicas de tu aplicación y la naturaleza de los datos que estás manejando. La optimización del rendimiento es un proceso iterativo, y es recomendable medir y perfilar tu aplicación para identificar las áreas que necesitan mejoras.