Managing Relationships in SQLAlchemy Data Models

Using the SQLAlchemy ORM to build data models with meaningful relationships.

Todd Birchard Follow
Jul 11, 2019 · 8 min read ★

We're going to be looking at vanilla SQLAlchemy: if you're looking for a guide on how to implement SQLAlchemy data models in Flask, it's probably worth checking out

There are plenty of good reasons to use SQLAlchemy, from managing database connections, to easy integrations with libraries such as Pandas. If you're in the app-building business, I'd be willing to bet that managing your app's data via an ORM is at the top of your list of use cases for SQLAlchemy.

Most software engineers likely find database model management to be easier than SQL queries. For people with heavy data backgrounds (like us), the added abstractions can be a bit off-putting: why do we *need* foreign keys to execute JOINs between two tables? Why do we need to distinguish when tables will have a **one-to-many** relationship, as opposed to a **many-to-many** relationship? These things aren't limitations of SQL, so what's with all the "extra work"?

The point of using an ORM is to result in *less* work for people building applications by translating database concepts into easily reproducible code in our app. Today we'll be checking out defining SQLAlchemy data models, with special attention to managing table relationships.

NOTE: We're going to be looking at vanilla SQLAlchemy: if you're looking for a guide on how to implement SQLAlchemy data models in Flask, it's probably worth checking out this post after you're done here.

Basic Model Definition

Before jumping into defining relationships, let's recap how to define a basic database model.

SQLAlchemy database models are classes which extend an SQLAlchemy base (such as a `declarive_base()`) which we import from the `sqlalchemy` package. Database models consist of **Columns**, which are data types unique to `sqlalchemy`:

```
from sqlalchemy import Column, Integer, String, Text, DateTime,
Float, Boolean, PickleType
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class ExampleModel(Base):
    """Data model example."""
    __tablename__ = "example_table"
    __table_args__ = {"schema": "example"}

id = Column(Integer,
                primary_key=True,
                nullable=False)
    name = Column(String(100),
                  nullable=False)
```

```
    description = Column(Text,
                         nullable=True)
    join_date = Column(DateTime,
                       nullable=False)
    vip = Column(Boolean,
                 nullable=False)
    number = Column(Float,
                    nullable=False)
    data = Column(PickleType,
                  nullable=False)

def __repr__(self):
        return '<Example model {}>'.format(self.id)
```

The above creates a model which utilizes each column type available to us via **sqlalchemy**. Each database model we create corresponds with a table, where each **Column** object in our model represents a column in the resulting table. When our app initializes SQLAlchemy, SQLAlchemy will create tables in our database to match each model (assuming one doesn't already exist).

We also set a couple of optional built-in variables in our model. `__tablename__` determines the name of the resulting database table, and `__table_args__` allows us to set which Postgres schema our table will live in (if applicable).

Creating Tables From Our Models

With our model created, we need to explicitly create the resulting table in our database. We do this by calling `create_all()` on the **Base** object, which is the object our model extends. Here's a quick script that takes care of this for us:

```
from models import Base
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from os import environ

# Create engine
db_uri = environ.get('SQLALCHEMY_DATABASE_URI')
engine = create_engine(db_uri, echo=True)

# Create All Tables
Base.metadata.create_all(engine)
```

I saved our `ExampleModel` class to a file called **models.py**, which is where we import **Base** from. Once we create our engine, the line `Base.metadata.create_all(engine)` then creates all tables associated with our models.

Check out the SQL query that resulted from our example model:

```
-- Sequence and defined type
CREATE SEQUENCE IF NOT EXISTS example.example_table_id_seq;

-- Table Definition
CREATE TABLE "example"."example_table" (
    "id" int4 NOT NULL DEFAULT
nextval('example.example_table_id_seq'::regclass),
    "name" varchar(100) NOT NULL,
    "description" text,
    "join_date" timestamp NOT NULL,
    "vip" bool NOT NULL,
    "number" float8 NOT NULL,
    "data" float8 NOT NULL,
    PRIMARY KEY ("id")
);
```

BONUS: Adding a Record

I know we've covered this, but *just in case:* here's how we'd add an instance of our `ExampleModel`:

```
newModel = ExampleModel(name='todd',
                        description='im testing this',
                        vip=True,
                        join_date=datetime.now())
session.add(newModel)
session.commit()
print(newModel)
```

This script should return `<Example model 1>`, because we added a `__repr__` method to our data model class:

One-to-Many & Many-to-One Relationships

One-to-many (or many-to-one) relationships are perhaps the most common type of database relationships. Examples of these include a *customers + orders* relationship (where single customers have multiple orders), or a *player + team* relationship (where a sportsball player belongs to a single team). We're going to build a couple of models to help demonstrate the latter.

I'm about to post a code snippet below which probably isn't going to make a lot of sense at first glance. *This is okay*- none of this comes to anybody naturally. We'll walk through this together:

```
from sqlalchemy import Column, Integer, String, Text, DateTime,
Float, Boolean, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class PlayerModel(Base):
    """Data model for players."""
    __tablename__ = "sqlalchemy_tutorial_players"
    __table_args__ = {"schema": "example"}

id = Column(Integer,
                primary_key=True,
                nullable=False)
    team_id = Column(Integer,

ForeignKey('example.sqlalchemy_tutorial_teams.id'),
                     nullable=False)
    name = Column(String(100),
                  nullable=False)
    position = Column(String(100),
                      nullable=False)

# Relationships
    team = relationship("TeamModel")

def __repr__(self):
        return '<Person model {}>'.format(self.id)

class TeamModel(Base):
    """Data model for people."""
    __tablename__ = "sqlalchemy_tutorial_teams"
    __table_args__ = {"schema": "example"}
```

```
    id = Column(Integer,
                    primary_key=True,
                    nullable=False)
        name = Column(String(100),
                      nullable=False)
        city = Column(String(100),
                      nullable=False)

    def __repr__(self):
            return '<Team model {}>'.format(self.id)
```

Off the bat there are a few things we *do* recognize. We have two models consisting of `Column`s: one model for *players,* and another model for *teams.* There are two additions here that are new.

First, we have the concept of **Foreign keys** (set on `PlayerModel`'s `team_id` column). If you're familiar with SQL, you should be good-to-go here. If not, think of it this way: a foreign key is a property of a column. When a foreign key is present, we're saying that this particular column denotes a *relationship* between tables: most common items of one table "belong" to items of another table, like when *customers* "own" *orders,* or when *teams* "own" *players.* In our example, we're saying that each *player* has a *team* as specified by their `team_id`. This way, we can marry data between our *players* table and our *team* table.

The other new concept here is **relationships.** Relationships complement foreign keys, and are a way of telling our application(not our database) that we're building relationships between two models. Notice how the value of our foreign key is `'example.sqlalchemy_tutorial_teams.id'`: **example** is our Postgres schema, and **sqlalchemy_tutorial_teams** is table name for our *teams* table. Compare this to the value we pass to our **relationship**, which is `"TeamModel"`: the class name of the target data model (not the table name!). Foreign keys tell *SQL* which relationships we're building, and relationships tell our *app* which relationships we're building. We need to do both.

The point of all this is the ability to easily perform JOINs in our app. When using an ORM, we wouldn't be able to say "join this model with that model", because our app would have no idea which columns to join *on*. When our relationships are specified in our models, we can do things like join two tables together without specifying any further detail: SQLAlchemy will know how to join tables/models by looking at what we set in

our data models (as enforced by the foreign keys & relationships we set). We're really just saving ourselves the burden of dealing with data-related logic while creating our app's business logic by defining relationships upfront.

PROTIP: SQLAlchemy only creates tables from data models if the tables don't already exist. In other words, if we have faulty relationships the first time we run our app, the error messages will persist the second time we run our app, even if we think we've fixed the problem. To deal with strange error messages, try deleting your SQL tables before running your app again whenever making changes to a model.

Back References

Specifying relationships on a data model allows us to access properties of the joined model via a property on the original model. If we were to join our **PlayerModel** with our **TeamModel**, we'd be able to access properties of a player's team via `PlayerModel.team.name`, where `team` is the name of our relationship, and `name` is a property of the associated model.

Relationships created in this way are one-directional, in that we can access team details through a player, but *can't* access player details from a team. We can solve this easily by setting a **back reference**.

When creating a relationship, we can pass an attribute called **backref** to make a relationship bi-directional. Here's how we'd modify the relationship we set previously:

```
# Relationships
team = relationship("TeamModel", backref="player")
```

With a **backref** present, we can now access player details of a team by calling `TeamModel.player`.

Performing a JOIN

Once you've successfully implemented a relationship between two data models, the best way to check your work is to perform a JOIN on these models. We won't waste time going into creating advanced SQLAlchemy ORM queries here, but at least we can check our work:

```
def join_example():
    records = session.query(PlayerModel).\
     join(TeamModel, TeamModel.id == PlayerModel.team_id).all()
    for record in records:
        recordObject = {'name': record.name,
                        'position': record.position,
                        'team_name': record.team.name,
                        'team_city': record.team.city}
        print(recordObject)
```

Here we JOIN **TeamModel** on **PlayerModel**, so we reference everything as a property of **PlayerModel**. Here's what this outputs with sample data:

```
{'name': 'Joe McMan',
 'position': 'Quarterback',
 'team_name': 'The Piggers',
 'team_city': 'Austin, TX'}
```

Many-to-Many Relationships

Setting foreign key relationships serve us well when we're expecting a table in our relationship to only have a single record per multiple records in another table (ie: one player per team). What if players could belong to *multiple* teams? This is where things get complicated.

As you might've guessed, many-to-many relationships happen between tables where n number of records from table 1 could be associated with n number of records from table 2. SQLAlchemy achieves relationships like these via *association tables.* An association table is a SQL table created for the sole purpose of explaining these relationships, and we're going to build one.

Check out how we define the **association_table** variable below:

```
from sqlalchemy import Column, Integer, String, ForeignKey, Table
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

```
association_table = Table('association', Base.metadata,
    Column('team_id', Integer,
ForeignKey('example.sqlalchemy_tutorial_players.team_id')),
    Column('id', Integer,
ForeignKey('example.sqlalchemy_tutorial_teams.id'))
)

class PlayerModel(Base):
    """Data model for players."""
    __tablename__ = "sqlalchemy_tutorial_players"
    __table_args__ = {"schema": "example"}

id = Column(Integer,
                primary_key=True,
                unique=True,
                nullable=False)
    team_id = Column(Integer,

ForeignKey('example.sqlalchemy_tutorial_teams.id'),
                     nullable=False)
    name = Column(String(100),
                  nullable=False)
    position = Column(String(100),
                      nullable=False)

# Relationships
    teams = relationship("TeamModel",
                         secondary=association_table,
                         backref="player")

def __repr__(self):
        return '<Player model {}>'.format(self.id)

class TeamModel(Base):
    """Data model for people."""
    __tablename__ = "sqlalchemy_tutorial_teams"
    __table_args__ = {"schema": "example"}

id = Column(Integer,
                primary_key=True,
                unique=True,
                nullable=False)
    name = Column(String(100),
                  nullable=False)
    city = Column(String(100),
                  nullable=False)

def __repr__(self):
        return '<Team model {}>'.format(self.id)
```

We're using a new data type **Table** to define a table which builds a many-to-many association. The first parameter we pass is the name of the resulting table, which we name `association`. Next, we pass `Base.metadata` to associate our table with the same declarative base that our data models extend. Lastly, we create two columns which serve as foreign keys to each of the tables we're associating: we're linking **PlayerModel**'s **team_id** column with **TeamModel**'s **id** column.

The essence of we're really doing here is creating a third table which associates our two tables. We could also achieve this by creating a third data model, but creating an association table is a bit more straightforward. From here on out, we can now query **association_table** directly to get records from our players and teams table.

The final step of implementing an association table is to sett a relationship on our data model. Notice how we set a **relationship** on **PlayerModel** like we did previously, but this time we set the **secondary** attribute equal to the name of our association table.

. . .

Originally published at https://hackersandslackers.com on July 11, 2019.

Programming Python Data Sql Software Development