Previous Next

Adding Relationships

Estimated Time: 5-6 hours

In the previous two assignments you created and worked with SQLAlchemy models for users, items, and bids for an auction site. But there is still an important component missing from the database: the **relationships** between the different parts of the system. For example, you don't know who owns each item, or to whom the bids belong. In this assignment you will be seeing how to create three different types of relationships between SQLAlchemy models.

Relationship types

Before looking at the code to create the relationships you need to understand what the three types of relationships are.

One-to-One

A one-to-one relationship links a single row of one table to a single row of another. For example, imagine you have a table which holds people, and a table which holds passport information. Each person is only associated with one piece of passport information. And each piece of passport information is only associated with one person. This is an example of a one-to-one relationship.

One-to-Many

A one-to-many relationship links a single row of one table to multiple rows of another. For example, think about a guitar and its manufacturer. Each guitar will only have one manufacturer. But a manufacturer will have created multiple guitars. So manufacturers have a one-to-many relationship to guitars. Or conversely guitars have a many-to-one relationship to manufacturers.

Many-to-Many

A many-to-many relationship links multiple rows of one table to multiple rows of another table. For example think about a pizza and its toppings. A pizza can have multiple toppings. And a single topping can be on multiple pizzas. This is an example of a many-to-many relationship.

Try It!

Try classifying the following relationships as either one-to-one, one-to-many, or many-to-many:

- Thinkful mentors and their students
- People and their spouses
- Films and the critics who have reviewed them
- TBay items and the users selling the item
- TBay items and the users who have bid on them
- TBay items and the bids placed on them

Creating the relationships

Now that you understand the differences between the types of relationship its time to see how to create the relationships between the tables using SQLAlchemy. Note that in order to run the examples in this section you will need to have a SQLAlchemy engine, session, and declarative base already initialized.

One-to-One

Take a look at the following example of how to create a one-to-one relationship between a person and their passport information:

```
from sqlalchemy import Column, Integer, String, Date, ForeignKey
from sqlalchemy.orm import relationship

class Person(Base):
    __tablename__ = 'person'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)

    passport = relationship("Passport", uselist=False, backref="owner"

class Passport(Base):
    __tablename__ = 'passport'
    id = Column(Integer, primary_key=True)
    issue_date = Column(Date, nullable=False, default=datetime.utcnow)

    owner_id = Column(Integer, ForeignKey('person.id'), nullable=False

beyonce = Person(name="Beyonce Knowles")
passport = Passport()
beyonce.passport = passport

session.add(beyonce)
session.commit()

print(beyonce.passport.issue_date)
print(passport.owner.name)
```

Here you create two models, one for a person, and one for their passport information. The person has a name field, and the passport information

contains an issue date.

Now look at the relationship. This is defined in two parts. In the `Passport` model there is an integer field called `owner_id` which will hold the id of a person. To ensure that the id refers to an existing row in the person table, the field has a `ForeignKey` constraint. The foreign key constraint specifies that a matching value must exist in a specified column of a different table. In this case the owner ID must be a value which exists in `'person.id'`; the id column of the person table.

The second part of the relationship is the `passport` field in the `Person` model. The most important part of the code is this: `passport = relationship("Passport")`. This says that there is a relationship between the Person model and the Passport model. SQLAlchemy is able to automatically work out that the two models are connected via the foreign key field in the Passport model.

With this relationship in place you can treat the `passport` field as if it were a regular part of the Person model. Notice how in the example you can access the passport like any other field through `beyonce.passport`.

The other two arguments passed to the `relationship` function also have a role to play. The `uselist=False` argument says that the passport should be treated as a single object rather than a list of objects. This is what guarantees that this is a one-to-one relationship - each person only has one set of passport information.

The `backref="owner"` says that as well as being able to access the passport information from the person through `person.passport`, you can also do the inverse and access the person through the passport object through `passport.owner`. This isn't strictly necessary, although it is pretty convenient.

One-to-Many

A one-to-many relationship is constructed in the same way as a one-to-one relationship except that the `uselist=False` argument is omitted. For example, here's a one-to-many relationship between manufacturer and the guitars they produce:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship

class Manufacturer(Base):
    __tablename__ = 'manufacturer'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    guitars = relationship("Guitar", backref="manufacturer")

class Guitar(Base):
    __tablename__ = 'guitar'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)

    manufacturer_id = Column(Integer, ForeignKey('manufacturer.id'),
                             nullable=False)

fender = Manufacturer(name="Fender")
strat = Guitar(name="Stratocaster", manufacturer=fender)
tele = Guitar(name="Telecaster")
fender.guitars.append(tele)

session.add_all([fender, strat, tele])
session.commit()

for guitar in fender.guitars:
    print(guitar.name)
print(tele.manufacturer.name)
```

Once again a foreign key relationship is set up between the `Guitar` and `Manufacturer` models. And again you set up a relationship with a backref, this time leaving out the `uselist=False` argument. This indicates that each manufacturer can build multiple guitars, and so the `guitars` field should be treated as a list.

With the models set up, you can then see how this works in practice. For the Stratocaster we use the backref to set the manufacturer field of the guitar. In contrast for the Telecaster we append the guitar to the `fender.guitars` list. Both of these methods have the same result - notice how the Stratocaster has been automatically appended to `fender.guitars`, and the Telecaster has had the `tele.manufacturer` field filled in automatically.

Many-to-Many

A many-to-many relationship works like two one-to-many relationships to an intermediate table. The table will hold foreign keys to columns on both sides of the relationship. For example, here's a many-to-many relationship between pizzas and toppings:

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

pizza_topping_table = Table('pizza_topping_association', Base.metadata
    Column('pizza_id', Integer, ForeignKey('pizza.id')),
    Column('topping_id', Integer, ForeignKey('topping.id'))
)

class Pizza(Base):
    __tablename__ = 'pizza'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    toppings = relationship("Topping", secondary="pizza_topping_associ
                            backref="pizzas")

class Topping(Base):
    __tablename__ = 'topping'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)

peppers = Topping(name="Peppers")
garlic = Topping(name="Garlic")
chilli = Topping(name="Chilli")

spicy_pepper = Pizza(name="Spicy Pepper")
spicy_pepper.toppings = [peppers, chilli]
```

```
vampire_weekend = Pizza(name="Vampire Weekend")
vampire_weekend.toppings = [garlic, chilli]


session.add_all([garlic, peppers, chilli, spicy_pepper, vampire_weeken
session.commit()

for topping in vampire_weekend.toppings:
    print(topping.name)

for pizza in chilli.pizzas:
    print(pizza.name)
```

First an association table is created to hold the foreign keys to the pizza and topping tables. A slightly different syntax from the normal declarative pattern is used to create this table, but fundamentally the same thing is happening - you are creating a table with two integer foreign key columns.

There are two models. The `Pizza` table contains a relationship to the `Topping` table with a backref as usual. The difference here is that a `secondary` argument is also supplied, to instruct SQLAlchemy to use the association table as an intermediary table. The backref will also automatically use this table in the same way.

Notice how in the example both the toppings on a pizza (`pizza.toppings`), and the pizzas which contain a topping (`topping.pizzas`) can be treated as list.

Try it!

Extend your *tbay.py* file to model the relationships between your User, Bid and Item classes. The models should be related according to the following rules:

- Users should be able to auction multiple items
- Users should be able to bid on multiple items

- Multiple users should be able to place a bid on an item

You will need to delete and recreate your database before making the changes to the models. To do that you can run: `dropdb tbay && createdb tbay` .

When you are happy with your classes, test them out by trying to:

- Add three users to the database
- Make one user auction a baseball
- Have each other user place two bids on the baseball
- Perform a query to find out which user placed the highest bid

When your TBay application is complete, push it up to GitHub and share it with your mentor. In the next unit you will be applying your skills with Postgres and SQLAlchemy, along with a new module, Flask, to create fully-functioning web-applications.

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

Mark as completed

Previous Next