Database Layer Relationships

Contents

Overview

Relationships between tables can be defined using the `db.relationship()` method.

One-to-Many Relationships

One-to-many relationships are one of the more common in databases.

We use the `db.relationship()` function to declare the relationship. However, we must use the `ForeignKey` class to separately declare the foreign key in the related table.

Consider this scenario where we have tables for individuals and the cars they own.

The `Cars` table is a collection of the individual cars, and the `Owners` table is the collection of the car owners. An `owner` has many `cars`, and a `car` has only one `owner`, making this a one-to-many relationship.

We define this relationship in the `Owners` class like so:

```
class Owners(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(30), nullable=False)
    last_name = db.Column(db.String(30), nullable=False)
    cars = db.relationship('Car', backref='ownerbr')
```

Here, `db.relationship()` creates an attribute `cars` that references all cars the owner is related to, i.e. any car whose `owner_id` value is equal to this owner's primary key.

We complete the relationship by assigning the primary key from `Owner` (`id`) as a foreign key in `Car`. This is declared using `db.ForeignKey()`.

```
class Cars(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    number_plate = db.Column(db.String(7), nullable=False)
    owner_id = db.Column(db.Integer, db.ForeignKey('owner.id'), nullable=False)
```

Using the Relationship Attribute

Defining a relationship attribute gives us a lot of handy options when creating or querying database entities.

Consider the following code:

```
# instantiate an owner
tiffany = Owners(first_name='Tiffany', last_name='Parker')

# instantiate two cars with Tiffany as the owner
car1 = Cars(number_plate="1234567", owner=tiffany)
car2 = Cars(number_plate="7654321", owner=tiffany)

# print a list of cars that belong to tiffany
print(tiffany.cars) # --> [<Cars 1>, <Cars 2>]

# print car owner's name using the backref
print(car1.ownerbr.first_name, car2.ownerbr.last_name) # --> Tiffany Parker
```

After creating our owner 'Tiffany', we can get a list of all the `Cars` objects that Tiffany is the owner of by using the `tiffany.cars` attribute. This means we can get her cars' information without having to query the `Cars` table directly.

Furthermore, the `backref='owner'` argument in the `Owners` table creates an `owner` attribute in the `Cars` class. Just as we could get Tiffany's cars with `tiffany.cars`, we can get car 1's owner with `car1.owner`.

The backref also allows us to pass through an `Owners` object as an argument when instantiating the `Cars` objects to model the relationship, rather than having to retrieve the owner's `id` primary key. This makes our code much more readable.

Many-to-Many Relationships

Consider a relationship between students and the classes they are enrolled in. We have a `Students` table and a `Classes` table.

Just like with one-to-many relationships, we use `db.relationship()` to declare the relationship in the parent table (in this case, `students`).

```
class Students(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(30), nullable=False)
    last_name = db.Column(db.String(30), nullable=False)
    enrolments = db.relationship('Enrolments', backref='students')

class Classes(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    subject = db.Column(db.String(30), nullable=False)
    enrolments = db.relationship('Enrolments', backref='class')
```

To implement the many-to-many relationship between them, an association table is required. In this case, we can name this table `Enrolments`.

`Enrolments` will consist of two foreign keys:

- `student_id` referring to `Students.id`.
- `classes_id` referring to `Classes.id`.

```
class Enrolments(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    class_id = db.Column('classes_id', db.Integer, db.ForeignKey('classes.id'))
    student_id = db.Column('student_id', db.Integer,
db.ForeignKey('students.id'))
    enrollment_date = db.Column('Date', db.String(30))
```

Tutorial

In this tutorial, we will implement a one-to-many relationship between two tables for a simple app.

Consider two tables, one contains countries and the other contains cities. Each country has many cities, but each city belongs to one country.

Setup

This tutorial assumes you are working on an Ubuntu VM, at least version `18.04 LTS`.

First, install apt dependencies:

```
sudo apt update
sudo apt install python3 python3-venv python3-pip
```

Create a directory named `flask-db-relationships` and make it your current working directory:

```
mkdir flask-db-relationships && cd $_
```

We now need to create a Python virtual environment to install our pip requirements in. Create a new virtual environment named `venv` and activate it:

```
python3 -m venv venv
source venv/bin/activate
```

Next, create three files named `app.py`, `create.py` and `requirements.txt`:

```
touch app.py create.py requirements.txt
```

Paste the following into `requirements.txt`:

```
flask
flask_sqlalchemy
pymysql
```

This is the list of pip dependencies that the app requires in order to run. Run the command to install them:

```
pip3 install -r requirements.txt
```

Creating the App

In the `app.py` file, paste in the following code to create our Flask object:

```
# Import everything we need
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
import os

app = Flask(__name__) # Declare Flask object

app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv("DATABASE_URI") # Set the
connection string to connect to a database
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app) # Declare SQLAlchemy object
```

Underneath the above code, define our table classes:

```
class Countries(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(30), nullable=False)
    cities = db.relationship('Cities', backref='country')

class Cities(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(30), nullable=False)
    country_id = db.Column(db.Integer, db.ForeignKey('countries.id'),
nullable=False)
```

Finally, add this line to start the app running:

```
if __name__=='__main__':
    app.run(debug==True, host='0.0.0.0')
```

We then need a `create.py` to create our schema.

```
from app import db, Countries, Cities

db.create_all() # Creates all table classes defined

uk = Countries(name = 'United Kingdom') #Add example to countries table
db.session.add(uk)
db.session.commit()

# Here we reference the country that london belongs to useing 'country', this is
what we named the backref variable in db.relationship()
ldn = Cities(name='London', country = uk)
mcr = Cities(name='Manchester', country = Countries.query.filter_by(name='United
Kingdom').first())

db.session.add(ldn)
db.session.add(mcr)
db.session.commit()

print(f"Cities in the UK are: {uk.cities[0].name}, {uk.cities[1].name}")
print(f"London's country is: {ldn.country.name}")
print(f"Manchester's country is: {ldn.country.name}")
```

For both `ldn` and `mcr`, `uk` is being assigned to the `country` parameter. However, for `mcr` we queried the table as if the Countries object had not been created in this session. Either approach works, but the `ldn` approach is easier to read.

We are then printing the information about the countries/cities in the database using our relationship attributes.

Completed `app.py`:

```


from flask import Flask
from flask_sqlalchemy import SQLAlchemy
import os

app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv("DATABASE_URI") # Set the
connection string to connect to the database
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

class Countries(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(30), nullable=False)
    cities = db.relationship('Cities', backref='country')

class Cities(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(30), nullable=False)
    country_id = db.Column(db.Integer, db.ForeignKey('countries.id'),
nullable=False)

if __name__=='__main__':
    app.run(debug==True, host='0.0.0.0')
```

Completed `create.py`:

```
from app import db, Countries, Cities

db.create_all() # Creates all table classes defined

uk = Countries(name = 'United Kingdom')
db.session.add(uk)
db.session.commit()

ldn = Cities(name='London', country = uk)
mcr = Cities(name='Manchester', country = Country.query.filter_by(name='United
Kingdom').first())

db.session.add(ldn)
db.session.add(mcr)
db.session.commit()

print(f"Cities in the UK are: {uk.cities[0].name}, {uk.cities[1].name}")
print(f"London's country is: {ldn.country.name}")
print(f"Manchester's country is: {ldn.country.name}")
```

Run the App

Because we are using an environment variable to define our database URI, we need to set it on the command line. Run the following command, replacing `<user>`, `<password>` and `<host_ip>` with the information relevant to your database:

```
export DATABASE_URI=mysql+pymysql://<user>:<password>@<host_ip>/testdb
```

Alternatively, if you don't have a separate MySQL server, you could use an sqlite database:

```
export DATABASE_URI=sqlite:///data.db
```

Next, run `create.py` to generate the table schema:

```
python3 create.py
```

You should see the following printed to the console:

```
Cities in the UK are: London, Manchester
London's country is: United Kingdom
Manchester's country is: United Kingdom
```

Clean Up

To stop your Flask application running, navigate back to your terminal and press `Ctrl+C`. You should now have control over your terminal again.

To deactivate the virtual environment, run:

```
deactivate
```

If you wish to delete the virtual environment, run:

```
rm -rf venv
```

Exercises

Model a many-to-many relationship between an `orders` table and a `products` table such that a single order can have many products and a product can be associated with many orders.

▶ Hint