04 Cookbook

April 4, 2021

0.1 Hybrid Attributes

- Hybird attributes are those that exhibit one behavior when accessed as a class method and another when accessed on an instance
- another way to think of this si that the attribute will generate valid `SQL` when it is used in a SQLAlchemy statement and when accessed on an instance the hybrid attribute will executr the python code directly against the instance
- the important point is that if we use the `hybird_property` or `hybrid_method` on a python object, we simply get a value »> Cookies.have_enough False
- if we use it in a query, it becomes a `SQL` statement session.query(Cookie).order_by(desc(Cookie.inventory_value))

[ ]:
```
def hybrid_attr():
    class Cookie(Base):
        __tablename__ = 'cookies'
        cookie_id = Column(Integer, primary_key=True)

    @hybrid_property
    def inventroy_value(self):
        return self.unit_cost * self.quantity

    # creates hybrid method because we need an idditional input
    @hybrid_method
    def bake_more(self, min_quantity):
        return self.quantity < min_quantity
```

0.2 Associateion Proxy

- an `association proxy` is a pointer across a relationship to a specific attribute
- it makes it easier to access an attribute across a relationship in code
- it would come in handly if we wanted a list of ingredient names that are used to make our cookie
- imagine if you had to do this every time you wanted ingredients: »> [ingredient.name for ingredient in cc_cookie.ingredients]
- we can turn that entire list comprehesion into »> cc_cookie.ingredient_names
- we can even add new ingredients through the association proxy »> cookie.ingredient_names.append('tamato') »> session.flush()

- be careful if you already had 'tamato' in the ingrdients, then the assocaition proxy will try to create it and you would get an error

To establish an association proxy that we can use for attribute access and ingredient creation, we need to do three things - Import the association proxy - Add an `__init__` method to the target object that makes it easy to create new instances with just the required values - Create an association proxy that targets the table name and column you want to proxy

[2]:
```
def association_proxy():

    from sqlalchemy.ext.associationproxy import association_proxy

    '''
    ...

    '''
    class Ingredient(Base):
        __tablename__ = 'ingredients'

        # defining an __init__ method that only requires a name
        def __init__(self, name):
            self.name = name

    class Cookie(Base):
        __tablename__ = 'cookies'


        # establish an association proxy to the ingredients name
        # attribute that we can refrence as ingredient_names
        ingredient_names = association_proxy('ingredients', 'name')
```

0.3 Integrating SQLAlchemy with Flask

- flask `SQLAlchemy` will provide preconfigured scoped sessions that afre tied to the page life cycle of your Flask application
  - `pip install flask-sqlalchemy`
- it is recommended to use the app factory pattern
- the app factory pattern uses a function that assembles an application with all the appropriate add-ons and configurations

[ ]:
```
def app_factory():

    from flask import Flask
    from flask.ext.sqlalchemy import SQLAlchemy
    from config import config

    db = SQLAlchemy()
```

```python
# defines the create_app app factory
def create_app(config_name):
    app = Flask(__name__)
    app.config.from_object(config[config_name])

    # initializes the instance with the app context
    db.init_app(app)
    return app
```

- only major change is that the sessions is nested in the `db.session` object
- one thing `Flask-SQLAlchemy` adds that is not found in normal is the the addition of a `query` method to every `ORM` data class
- it is not reccomended you use it
- it looks like the following
  - `Cookie.query.all()`

0.4 SQLAcodegen

- `SQLAcodegen` uses reflection to build a collection of OREM data classes that you can use in your application code base to avoid reflecting the database multiple times
- `SQLAcodegen` has the ability to detect `many-to-one`, `one-to-one`, and `many-to-many` relationships

- first install `SQLAcodegen`
  - `pip install sqlacodegen`
- to run SQLAcodegen we need to specifiy a database connection string for it to connect to
  - `sqlacodegen sqlite:///Chinook_Sqlite.sqlite`
- when we run the command below, it builds up a complete file that contains all the ORM data classes of the database along with the proper imports
  - 'sqlacodegen sqlite://Chinook_Sqlite.sqlite –tables Atrist, Track
- the file is ready for use in our application
- you might need to tweak the setting for the `Base` object if it was established elsewhere

- when we speicfied the `Artist` and `Track` tables, `SQLAcodegen` built classes for those tables and all the tbales had a relationship with those tables
- `SQLAcodegen` does this to ensure that the code it builds is ready to be used
- if you want to save the generated classes directly to a file, you can do so with standard redirection
  - `sqlacodegen sqlite:///Chinook_Sqlite.sqlite --tables Artist,Track > db.py`