Database Layer Schema

Contents

Overview

In this module, we will implement database tables for an app using SQLALchemy.

Models

One of the main reasons for using SQLAlchemy is that it eliminates the need to log into a MySQL console to create and manage tables. Instead, we use object relational mapping to treat tables and entities as Python classes and objects, respectively.

This is done by designing the models for the app's tables in Python. It is good practice to keep them abstracted from other parts of the app in their own file.

Declaring Models

The `SQLAlchemy` class provides us with the methods and classes we need to design our tables. Each table is declared as a class using the declarative base class `db.Model`.

```
from application import db # import the sqlalchemy object (db) created for our
app

class Person(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)
```

The `db.Column` function creates the column with parameters that can be set to configure options for that column.

Column options

This is a non-exhaustive list of commonly used parameters.

> Note: the examples below are not necessarily from the same conceptual database, and are just used to illustrate usage.

Column data types

Argument	Example	Description
db.Integer	`id = db.Column(db.Integer)`	This assigns the data type `integer` to the `id` column.
db.String(max_str_length)	`first_name = db.Column(db.String(30))`	This assigns the data type `string` with a maximum of 30 characters to the column `first_name`.
db.Boolean	`alive = db.Column(db.Boolean)`	This assigns the data type `boolean` to the `alive` column.
db.DateTime	`date = db.Column(db.DateTime)`	This assigns the data type `DateTime` to the `date` column.
db.Float	`height = db.Column(db.Float)`	This assigns the data type `float` to the `height` column.

Constraints

Argument	Example	Description
nullable	`first_name = db.Column(db.String(30), nullable = False)`	Sets whether or not this column can contain empty (null) values.
unique	`username = db.Column(db.String(30), unique = True)`	Each entry in the column `username` must be unique.
primary_key	`id = db.Column(db.Integer, primary_key=True)`	Assigns the column `id` to be the primary key of the table.
default	`alive = db.Column(db.Boolean, default=True)`	Sets the default value in the `alive` column to `True`.

Tutorial

This tutorial shows you how to create a table using an SQLAlchemy classes.

Prerequisites

Before starting this tutorial, you will need to create a MySQL server instance for your Flask application to connect to with a database named `testdb`.

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-schema` and make it your current working directory:

```
mkdir flask-db-schema && 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

Paste the following into `app.py`:

```
from flask import Flask # Import Flask class
from flask_sqlalchemy import SQLAlchemy # Import SQLAlchemy class
import os

app = Flask(__name__) # create Flask object

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

Here, we are instantiating the Flask app object, configuring the connection string by referencing an environment variable, and instantiating the SQLAlchemy object.

Next, we will create a `Users` class for our table in `app.py` under the above code:

```
class Users(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)

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

Next, paste the following code into `create.py`:

```
from app import db, Users

db.drop_all()
db.create_all()

testuser = Users(first_name='Grooty',last_name='Toot') # Extra: this section
populates the table with an example entry
db.session.add(testuser)
db.session.commit()
```

Here we are importing the SQLAlchemy object `db` and the `Users` class defined in `app.py`.

The two functions `db.drop_all()` and `db.create_all()` delete all tables then create all tables defined for our `db` object, allowing us to create our database schema from fresh based on how the classes have been defined.

The last three lines populate the `Users` table with a user entity.

Running 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
```

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

```
python3 create.py
```

You should now be able to log into your MySQL database and use a `SELECT` query to find the information we have just created.

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

Create another table that uses some of the data types and constraints which are shown above.