Create database

DESKTOP-AALCCGH:~$ sudo -u postgres psql

[sudo] password for xxx:

psql (14.8 (Ubuntu 14.8-0ubuntu0.22.04.1))

Type "help" for help.

postgres=#

- CREATE DATABASE estate WITH OWNER = postgres ENCODING = 'UTF8' CONNECTION LIMIT = -1;
- \c estate

Create tables see below

Here's the structure of your Flask project directory:

your_project/

├── app.py

├── templates/

│ ├── customers.html

│ ├── create_customer.html

│ ├── agents.html

│ ├── create_agent.html

│ ├── realestates.html

```
|   └── create_real_estate.html
└── ...
```

build a web application using Python with SQL Alchemy and Flask or Django, you can follow these steps:

Step 1: Set up your development environment.

Install Anaconda: Download and install Anaconda from the official website (https://www.anaconda.com/products/individual). Anaconda includes Python and useful packages for data science and web development.

Open a terminal or Anaconda Prompt to execute commands.

Step 2: Create a new virtual environment

Create a new virtual environment for your project by running the following command:

Copy code

conda create --name mywebapp

Activate the virtual environment:

On Windows: conda activate mywebapp

On macOS and Linux: source activate mywebapp

Step 3: Install dependencies

Install Flask or Django:

For Flask: Run pip install Flask

For Django: Run pip install Django

Install SQL Alchemy and the PostgreSQL driver:

pip install SQLAlchemy psycopg2

Step 4: Set up your project structure.

Create a new directory for your project and navigate to it in the terminal.

Use Flask or Django to generate the initial project structure:

For Flask: Run flask init

For Django: Run django-admin startproject mywebapp .

Step 5: Configure the database connection

Open the configuration file for your chosen framework:

For Flask: Modify the app.py or __init__.py file.

For Django: Modify the settings.py file in the mywebapp directory.

Configure the PostgreSQL database connection by specifying the database name, username, password, and host.

Step 6: Create models.

Define your database models using SQL Alchemy's ORM:

For Flask: Create a models.py file and define your models using SQL Alchemy's declarative syntax.

For Django: Define your models in the models.py file within the app directory.

Step 7: Create views/routes.

Define the routes or views for your application:

For Flask: Create routes in the app.py or views.py file.

For Django: Define views in the views.py file within the app directory.

Step 8: Create templates.

Create HTML templates for your web pages using Flask's or Django's template system.

For Flask: Create templates in a templates folder.

For Django: Create templates in the app directory's templates folder.

Step 9: Run the application.

Start the development server:

For Flask: Run flask run

For Django: Run python manage.py runserver

Access your web application in a browser by visiting the appropriate URL displayed in the terminal.

install in case we need to alter some things.

pip install Flask-Migrate

By following these steps, you should be able to create a web application using Python, SQL Alchemy, and either Flask or Django. Remember to adjust the code and configurations according to your specific requirements and project structure.

Step 1: Set up your development environment.

Install Anaconda: Download and install Anaconda from the official website (https://www.anaconda.com/products/individual). Anaconda includes Python and useful packages for data science and web development.

Open a terminal or Anaconda Prompt to execute commands.

Step 2: Create a new virtual environment.

Create a new virtual environment for your project by running the following command:

conda create --name mywebapp

Activate the virtual environment:

On Windows: conda activate mywebapp

On macOS and Linux: source activate mywebapp

Step 3: Install dependencies.

Install Flask and SQL Alchemy:

pip install Flask SQLAlchemy

Install the PostgreSQL driver:

pip install psycopg2

Step 4: Set up your project structure

Create a new directory for your project and navigate to it in the terminal.

Create a Python file named app.py as the entry point for your application.

Step 5: Configure the database connection.

Import the necessary modules in your app.py file:

python

from flask import Flask

from flask_sqlalchemy import SQLAlchemy

Configure the PostgreSQL database connection by specifying the database URI:

python

app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://username:password@localhost/mydatabase'

db = SQLAlchemy(app)

Replace username, password, and mydatabase with your PostgreSQL credentials.

Step 6: Define models.

Define your database models using SQL Alchemy's ORM. In your app.py file, add the following code:

python

Add any additional fields or relationships to the models based on your requirements.

Step 7: Create routes and views.

Define the routes and views for your application in the app.py file:

See app.py

```
@app.route('/')
def index():
    return 'Welcome to the Real Estate Application'

@app.route('/customers')
def customers():
    # Retrieve and display customer information
    # You can use the Customer model to query the database
    return 'Customer information page'

# Add more routes and views for other functionalities

if __name__ == '__main__':
    app.run()
```

Customize the routes and views based on the CRUD operations you want to perform.

Step 8: Run the application

Start the Flask development server by running the following command:

```
flask run
```

Access your web application in a browser by visiting the URL displayed in the terminal.

With these steps, you should have a web application using Python, SQL Alchemy, and Flask, and you can perform CRUD operations on the PostgreSQL database. Remember to adjust the code and configurations based on your specific requirements.

```
CREATE TABLE customer (
    customer_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    contact_details VARCHAR(100),
    account_balance NUMERIC(10, 2));

CREATE TABLE real_estate_agent (
    agent_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    contact_details VARCHAR(100));

CREATE TABLE real_estate (
    property_id SERIAL PRIMARY KEY,
    title VARCHAR(100),
    location VARCHAR(100),
    price NUMERIC(10, 2),
    listing_date DATE,
    sell_date DATE,
    customer_id INTEGER,
    FOREIGN KEY (customer_id) REFERENCES customer (customer_id));
```

Note I need to alter and migrate

ALTER TABLE real_estate ADD COLUMN agent_id INTEGER;

ALTER TABLE real_estate ALTER COLUMN sell_date DROP NOT NULL;

ALTER TABLE real_estate ALTER COLUMN sell_date SET DEFAULT NULL;

we can simply create

```
CREATE TABLE real_estate (
    property_id SERIAL PRIMARY KEY,
    title VARCHAR(100),
    location VARCHAR(100),
    price NUMERIC(10, 2),
    listing_date DATE,
    sell_date DATE,
    customer_id INTEGER,
    FOREIGN KEY (customer_id) REFERENCES customer (customer_id) ,agent_id INTEGER);
```

To enable CRUD operations for an agent on all tables (customer, real_estate_agent, and real_estate), you'll need to create appropriate routes and views in your Flask application. You implement CRUD functionality for each table:

Customer CRUD:

Create: Provide a form for the agent to add a new customer to the database.

Read: Display a list of all customers, showing their details.

Update: Allow the agent to edit the details of a specific customer.

Delete: Provide an option to delete a customer from the database.

Real Estate Agent CRUD:

Create: Provide a form for the agent to add a new real estate agent to the database.

Read: Display a list of all real estate agents, showing their details.

Update: Allow the agent to edit the details of a specific real estate agent.

Delete: Provide an option to delete a real estate agent from the database.

Real Estate CRUD:

Create: Provide a form for the agent to add a new real estate property to the database.

Read: Display a list of all real estate properties, showing their details.

Update: Allow the agent to edit the details of a specific real estate property.

Delete: Provide an option to delete a real estate property from the database.

To implement these functionalities, you can define routes and corresponding view functions in your app.py file. Here's an example for the Customer CRUD operations:

python

```
from flask import Flask, render_template, request, redirect

from flask_sqlalchemy import SQLAlchemy


app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://username:password@localhost/mydatabase'

db = SQLAlchemy(app)


# Customer model definition (as previously mentioned)


@app.route('/')
```

```
def index():
    return 'Welcome to the Real Estate Application'


# Customer CRUD routes
@app.route('/customers', methods=['GET'])
def customers():
    # Retrieve all customers from the database
    all_customers = Customer.query.all()
    return render_template('customers.html', customers=all_customers)


@app.route('/customers/create', methods=['GET', 'POST'])
def create_customer():
    if request.method == 'POST':
        # Get data from the submitted form
        name = request.form['name']
        contact_details = request.form['contact_details']
        account_balance = request.form['account_balance']


        # Create a new customer object
        new_customer = Customer(name=name, contact_details=contact_details,
account_balance=account_balance)


        # Add the new customer to the database
        db.session.add(new_customer)
        db.session.commit()
```

```
            return redirect('/customers')
        else:
            return render_template('create_customer.html')

# Add routes and view functions for update and delete operations

if __name__ == '__main__':
    app.run()
```

Similarly, you can create routes and view functions for the Real Estate Agent and Real Estate tables.

Make sure to create the corresponding HTML templates (customers.html, create_customer.html, etc.) to render the views and forms.

You can modify and expand upon this example to include the necessary CRUD operations for each table.

The migrate not needed if the tables created correctly

==========Work log==

Update the database schema to reflect the changes. If you are using Flask-Migrate, run the following commands in your command line:

```
pip install Flask-Migrate

flask db init

flask db migrate -m "Add agent_id column to real_estate"

flask db upgrade

flask db migrate -m "Initial migration"

flask db upgrade
```

http://127.0.0.1:5000/realestates

/customers

/agents

http://127.0.0.1:5000 (This will be home page)