Creating a students records system with SQLite and Flask

This assumes some basic familiarity with Flask and SQLite.

Create database and table

First, you need somewhere to save all your data. You can create an SQLite database from Python code as below. The program below creates a SQLite database ‘students.db ‘ where the student tables are created.

```
import sqlite3

conn = sqlite3.connect('students.db')
print("Opened database successfully");

conn.execute('CREATE TABLE students (name TEXT, addr TEXT,
city TEXT)')
print("Table created successfully");
conn.close()
```

Once the database is setup with the right fields, we can create a web app that uses it, import the things we need and create a new student function that enters some data. We do this in a new file. I’ve called mine main.

```
from flask import Flask, render_template

app= Flask(__name__)

@app.route('/enternew')
def new_student():
   return render_template('student.html')
```

The corresponding student.html file is

```
<html>
   <body>

      <form action = "{{ url_for('addrec') }}" method =
"POST">
         <h3>Student Information</h3>
         Name<br>
         <input type = "text" name = "nm" /></br>

         Address<br>
         <textarea name = "add" ></textarea><br>

         City<br>
         <input type = "text" name = "city" /><br>

         <input type = "submit" value = "submit" /><br>
      </form>

   </body>
</html>
```

Add record

As can be seen, the form data is published to the '/addrec' (standing for add record) endpoint mapped to the addrec () function.
This means that the addrec () function gets the html form's data from the POST method and inserts the student table. The message corresponding to the success or error in the insert operation will be rendered as 'result.html'. Below the function you created above, in the main file, add the following. Before you do however, you will need to add request into the list of things imported from flask.

```
@app.route('/addrec',methods = ['POST', 'GET'])
def addrec():
    if request.method == 'POST':
        try:
            name = request.form['name']
            addr = request.form['add']
            city = request.form['city']

            with sqlite3.connect("students.db") as con:
                cur = con.cursor()
                cur.execute("INSERT INTO students
(name,addr,city) VALUES (?,?,?)",(name,addr,city) )
                con.commit()
                msg = "Record successfully added"

        except:
            con.rollback()
            msg = "error in insert operation"
        finally:
            con.close()
            return render_template("result.html",msg = msg)
```

We also need to add a template for result.html this uses the output from the result operation to display it.

```
<!doctype html>
<html>
   <body>

      result of addition : {{ msg }}
      <h2><a href = "\">go back to home page</a></h2>

   </body>
</html>
```

List students in the db

One of the things we might need is a way to showcase all the students in the records system. We can put this in the /liststudents endpoint and run a query to get that info passing it to the html.

```
@app.route('/liststudents')
def listStudents():
   con = sqlite3.connect("students.db")
   con.row_factory = sqlite3.Row

   cur = con.cursor()
   cur.execute("select * from students")

   rows = cur.fetchall();
   return render_template("studentlist.html",rows = rows)
```

The next step is to write the html template for this, making use of the ‘rows’ data we bring in.

```
<!doctype html>
<html>
   <body>

      <table border = 1>
         <thead>
            <td>Name</td>
            <td>Address</td>
            <td>city</td>
         </thead>

         {% for row in rows %}
            <tr>
               <td>{{row["name"]}}</td>
               <td>{{row["addr"]}}</td>
               <td> {{ row["city"]}}</td>
            </tr>
         {% endfor %}
      </table>

      <a href = "/">Go back to home page</a>

   </body>
</html>
```

And finally, the home page function and html:

```
@app.route('/')
def home():
    return render_template('home.html')
```

HTML:

```
<html>
   <body>

    <table border = 1>
        <tr>
            <td><a href = "/liststudents">Show all
Students</a></td>
            <td><a href = "/enternew">Add new student</a></td>
        </tr>
        </table>
    <a href = "/">Go back to home page</a>
   </body>
</html>
```

Further Tasks

1) Add to the page to make it look nicer, with consistent styling and naming,

2) Give at least one other query and response including page setup (e.g. searching for students in a city or with a given name)

3) Expand the student records search so that more information is taken in, this will mean modifying the database too.

4) Improve the error handling for the user.

5) Add the ability to edit data within the flask application on students

6) Create a new page and function for courses

7) Link the courses to a student

Adding authentication and login

At the moment our app has no authentication at all, anyone can access it, instead, we'll make it so that only people that are logged in can access the private areas.

We'll do this using another database that handles all the login stuff and the use of a few flask libraries.
So, let's get the installation and setup out the way.

Run:

```
Pip install flask_bcrypt
Pip intall flask_login
```

Now, in order to test our ability to login we need to do some setup to create some credentials and a database to hold them. First, we create a new script to setup the database.
Create a new script, add to it the following;

```
import sqlite3

conn = sqlite3.connect('login.db')

conn.execute('create table login(user_id INTEGER PRIMARY KEY
AUTOINCREMENT,email text not null,password text not null);')
conn.execute('insert into login (user_id,email,password)
VALUES (120,"xyz@mail.com","123xyz")')
conn.execute('insert into login (email,password) VALUES
("abc@mail.com","123abc")')
conn.commit()
```

This creates a new database "login.db" and in that database creates a table with three fields- the primary key, the unique identifier for that record in the database. We set this as autoincrement so it adds 1 to the previous value of key- this means we also have a unique identifier. Then we have a email and password field. Both are text, both are not null- meaning they can't be empty.

Finally, we insert into the login tables some sample data, here we're not protecting the data, we're just using for testing.
YOU SHOULD NOT DO THIS FOR ANYTHING OTHER THAN TESTING.
It's a very insecure process- we'll go over protecting the data later.
Finally, we commit the changes to the database.

Run this script and it should create and populate your database.

We now have two sample database users to test our login page with.

Our next step is to write the login.html page we'll use to login, in keeping with prior pages, we'll keep it plain, no css and no base template. If you've completed the previous, you can add them on.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
</head>
<body>
<h3>Login</h3>
<br>
<div>
    {% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
      {{ message }}
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
</div>
<div>
        <form method="POST" action="/login">
                    <input type="email" name="email"
placeholder="Your Email" autofocus="">
                    <input type="password" name="password"
placeholder="Your Password">
                <label class="checkbox">
                    <input type="checkbox" name="rememberMe">
                    Remember me
                </label>
            <button class="button">Login</button>
        </form>
    </div>
</body>
</html>
```

This is simply a form that has boxes for email, password and rememberme, the email and password types are set and placeholders are set. On pressing the login button it'll send it via a POST request to the /login endpoint- which we'll write shortly .

Back to your app.py and, at the top of your app.py file, add the following (some of this may already be there- the bits in light blue are the newer bits.

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

```
from flask_login import LoginManager, UserMixin,
login_required, login_user, logout_user, current_user
import sqlite3
from flask_bcrypt import Bcrypt
```

Then, at the top of your application add the below

app= Flask(__name__) #this should already be here

```
app.config['SECRET_KEY'] = 'thisIsSecret'
login_manager = LoginManager(app)
login_manager.login_view="login"
```

Secret keys are used to manage sessions- keeping the data for a session- data across multiple request about the user and previous requests.

We then setup a flask login manager- giving it the app name and telling it what the view will be- the endpoint we're going to setup below the base login just returns the html page- we'll later use a post request too but for now:

```
@app.route('/login')
def login():
    return render_template('login.html')
```

Following that, we need to implement something that represents the user, flask-login uses the userMixin class to do this. We create a class that inherits from this, declaring the construction with the fields we want- in this case, ID, email and password. This can be done within your app.py main script.

```
#creates a user model represent ing the user
class User(UserMixin):
    def __init__(self, id, email, password):
         self.id = id
         self.email = email
         self.password = password
         self.authenticated = False
         def is_active(self):
            return self.is_active()
         def is_anonymous(self):
            return False
         def is_authenticated(self):
            return self.authenticated
         def is_active(self):
            return True
         def get_id(self):
            return self.id
```

Next, we need to add in the post method- I’ve separated it out, but you could just include it in login, adding an if post check like we do prior.

```
@app.route('/login', methods=['POST'])
     def login_post():
     #check if already logged in- if so send home
         if current_user.is_authenticated:
             return redirect(url_for('home'))
                # do the standard database stuff and find the user
     with email
         con = sqlite3.connect("login.db")
         curs = con.cursor()
         email= request.form['email']
         curs.execute("SELECT * FROM login where email = (?)",
     [email])
     #return the first matching user then pass the details to
     #create a User object – unless there is nothing returned then
     flash a message
         row=curs.fetchone()
         if row==None:
             flash('Please try logging in again')
             return render_template('login.html')
         user = list(row);
         liUser = User(int(user[0]),user[1],user[2])
         password = request.form['password']
         match = liUser.password==password
     #if our password matches- run the login_user method
         if match and email==liUser.email:
             login_user(liUser,remember=request.form.get('remember'
     ))
             redirect(url_for('home'))
         else:
             flash('Please try logging in again')
             return render_template('login.html')
         return render_template('home.html')
```

Eagle eyed amongst you will notice that there is a couple things here. First, login_user hasn’t been declared, it’s brought in from Flask and is the bit that tells Flask that a user is logged in. Second, the user of flash. Flash allows a message to quickly be added to the HTML providing the template already has a flash section defined. Looking back at the login.html template we do this using the following code which should already be in there.

```
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
      {{ message }}
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

In order for login_user to work we also need Flask to be able to load the user in a way in which it expects. We create a load_user method in the format below and tell flask login_manager to use this.

```
@login_manager.user_loader
def load_user(user_id):
   conn = sqlite3.connect('login.db')
   curs = conn.cursor()
   curs.execute("SELECT * from login where user_id =
(?)",[user_id])
   liUser = curs.fetchone()
   if liUser is None:
      return None
   else:
      return User(int(liUser[0]), liUser[1], liUser[2])
```

And that's it, a very basic login system, storing passwords in plaintext, in order to actually function and protect pages though we need to tell flask-login which bits to protect.
We do this using @login_required between the app route and the function name. E.g. in the below listStudents and new_student require login but login does not.

```
@app.route('/liststudents')
@login_required
def listStudents():

@app.route('/enternew')
@login_required
def new_student():

@app.route('/login', methods=['POST'])
def login_post():
```

We can also provide a method for people to logout too! We could (and should) link login and logout buttons and links to the navbar.

```
@app.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for('home'))
```

Protecting passwords

Plain text passwords in a database are vulnerable to interception and theft, instead of using plain text we're going to use a hashing and salting method.
Hashing is a process that ciphers text into something different using a specific algorithm, once done it's impossible to change back. Salting adds random characters into the mix before or after the password before it's hashed so the end result is even harder to decipher or use.

So, let's go through and add it in, we'll be using the popular bcyrpt library and it's flask companion. Add the import and then setup your app at the top.

```
from flask_bcrypt import Bcrypt

app= Flask(__name__) # should already be here
bcrypt = Bcrypt(app) #creates new bcrypt instance
```

We then need to make sure our login methods use this new encryption- any of our old passwords that are plain text will then stop working, we'll need users to create new ones with the hashing algorithm. We can either do that manually or instead implement a register page later to try it out. See the change below in login_post- I've ensured it's a different colour. Everything else stays the same.

```
@app.route('/login', methods=['POST'])
def login_post():
      …
    liUser = User(int(user[0]),user[1],user[2])

    password = request.form['password']
    match = bcrypt.check_password_hash(liUser.password,
password)

    if match and email==liUser.email:
      ….
```

Instead of just checking to see if the hashes match or the password matches, we now need to use the check_password_hash method, this checks to make sure that- when hashed in the same way, the password matches.

To our existing implementation that's all we need to change, but, we need a way to input and setup users.

Registering new users

Once again I've separated out the GET and POST requests, with the standard register just displaying the page, the POST

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

```
def register():
    return render_template('register.html')

@app.route('/register', methods=['POST'])
def register_post():
#check is authenticated/ logged in already
    if current_user.is_authenticated:
        return redirect(url_for('home'))
           #standard DB stuff
    con = sqlite3.connect("login.db")
    curs = con.cursor()
    email= request.form['email']
    password = request.form['password']
# we use bycrpt to hash the password we’ve put in with salt
    hashedPassword=bcrypt.generate_password_hash(password)
# then add it into the database- now hashed
    con.execute('insert into login (email,password) VALUES
(?,?)',[email,hashedPassword])
    con.commit()
    return render_template('home.html')
```

Finally, we need to add in the register.html page so that people can actually register. It’s very similar in structure to the login page.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Register</title>
</head>
<body>
<h3>Register</h3>
<br>
<div>
    {% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
    {% for message in messages %}
      {{ message }}
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
</div>
<div>
  <form method="post" action="/register">
    <input type="email" name="email" placeholder="Your Email"
autofocus="" required>
    <input type="password" name="password" placeholder="Your
Password" required>
    <button class="button">Register</button>
  </form>
    </div>
</body>
</html>
```