Flask Python (9) – Authentication

In the modern world websites generally have a splash page and then other links you can follow to find out more about the company or site. In order to target information at users (or indeed to keep some information from others) users can sign into a website. In this section we will look at how Flask helps to allow users to sign up, login and then go to the Library Resources page.

Login Link

Step 1: We need to add a link on the navigation bar so that users can login in. Create a link to a new page (login.html) at the top of the page.

Step 2: Create a route int *flask_app.py* which takes users to `login.html` when they click the link above. The route will need the GET and POST methods.

Step 3: We no longer need the link to the Library Resources page as that will be where the user goes when they login, so remove that link from the navigation bar.

Step 4: Also create a logout option on the Navigation bar which will log the users out. Again, you will need to create a route for this called `/logout.`

Login Class

Step 5: In *forms.py* create a LoginForm class to hold the data being passed from our login form. This just needs two fields email and password and then a submit button.

```
class LoginForm(FlaskForm):
    email = StringField('email')
    password = PasswordField('Password')
    submit = SubmitField('Log In')
```

Login Page.

Step 6: The *login.html* page is quite straight forward and similar to the SignUp page we created earlier:

```
{% extends "base.html" %}
{% block content %}
<div class="container w-50">
<div class="p-4 text-center">
<h2> Login</h2>
</div>
<form method="POST">
{{form.hidden_tag()}}
<div class="row p-2">
    <div class="col-sm-5">{{form.email.label}}</div>
    <div class="col-sm-5">{{form.email}}</div>
</div>
<div class="row p-2">
    <div class="col-sm-5">{{form.password.label}}</div>
```

```
            <div class="col-sm-5">{{form.password}}</div>
        </div>
        <div class="row p-3">
            {{form.submit}}
        </div>
            </form>
        </div>
{% endblock %}
```

Users Database

Step 7: Like we did with books we need somewhere to store users. In *models.py* create a function - `create_user_table()` - which will create a new table called T_users which has the following fields:

```
T_users = { email (primary key), FirstName, Surname, Password)
```

Step 8: Write a function `add_user(email, fname, sname, password)` which will allow you to populate the table with a couple of records of data using an INSERT query.

Step 9: Also, create a `get_user(email)` function which will return the email, FirstName, Surname and Password of a user from their email.

Step 10: Write another function `check_user_password (email, passw)` which will check that the users email and password match. If they do it returns `True` otherwise `False`.

Logging In

Step 11: When a user logs in (i.e. submits the LoginForm) we want to check the data in our form is valid and if it is we load the library page.

```
@app.route('/login',methods=["GET","POST"])
def login():
    form = LoginForm()
    if form.validate_on_submit():
       return redirect (url_for('library'))
    return render_template('login.html', form = form)
```

Examining this code we see that a form object is created using the `LoginForm()` function from *forms.py* (make sure you import it), if a valid form has been submitted we use the `redirect url_for()` statement to redirect our site the `/library` route, if not we reload the *login.html* page.

Validating a User

Step 12: So far, we have a login page which allows a user to login regardless of their email or password. Use the validation of the LoginForm class to check if the user exists and if their password is correct. Do this in the *form.py* file as in the previous section using form validation. The functions you wrote in the *models.py* file to interrogate the database will be useful here.

Step 11: If errors occur with the validation, they should appear on the login page so amend *login.html* to do this as we did in the previous section.

Login Manager

Users have now logged in, but it is still possible for anyone to access the *library.html* page by just typing in the route `/library`. We need something to make sure this page is only accessed by authenticated users.

Step 12: At the top of you *flask_app.py* file you will need to import some libraries from flask_login.

```
from flask_login import LoginManager, login_user, login_required, logout_user, current_user

from flask_login import UserMixin
```

These libraries will allow the creation of a session so we know if someone has signed in and who they are. Without this functionality the user would have to sign in every time they wanted to visit a page.

Step 13: After you have defined the app variable in flask_app.py paste in the following code.

```
login_manager = LoginManager()

login_manager.init_app(app)

login_manager.login_view = 'login'

## Login Manager initialisation

@login_manager.user_loader

def load_user(email):

    this_user = get_user(email)

    return User(this_user)

## A class to hold the details of the logged in user

class User(UserMixin):

    def __init__(self,n):

        self.id = n[0][0]

        self.fname = n[0][1]

        self.Surname = n[0][2]
```

We are using the `get_user(email)` function from *models.py* so make sure it is being imported. As this returns a list of tuples we will take the first one – as emails are unique there should just be one tuple! Also note that `UserMixin` requires the unique attribute to be "id". Even thought we are using the email address it must still be the label `id` and not `email`.

Step 14: We now need to amend our `login()` function:

```
@app.route('/login',methods=["GET","POST"])

def login():

    form  = LoginForm()

    if form.validate_on_submit():
```

```
            u = User(get_user(form.email.data))
            login_user(u)
            #flash ("Login succesful")
            return redirect (url_for('library'))
        return render_template('login.html',form = form)
```

We take our user (based on their email) and note that we have validated that the email and password are correct. So we now run the `login_user()` function which adds the details of the user to the session as being authenticated.

Step 15: Given that we now have a session running we can use this to only show the library page if there is an authenticated user in the session. This will check that there is an authenticated user and also retrieve the first name of the user for a greeting. You can add the {%else%} clause to display a message saying that the user is not authenticated.

```
{% if current_user.is_authenticated %}
Welcome {{current_user.fname}},
```

Logging out

Step 16: The logging out is very straightforward:

```
@app.route('/logout',methods=["GET","POST"])
def logout():
    logout_user()
    return redirect (url_for('index'))
```

Step 17: You can also return to the base.html file and make sure the user only sees a login option if there is no user and a logout option if there is – the same conditional is used.

SignUp Form

Step 18: You now have all the tools you require to get your signup form displaying, adding a new user if the details are correct. You will need the wtforms validator EqualTo to check that the passwords match on your sign up form.

Password Encryption

Obviously we do not want to be storing raw password in our database. There are various reasons for this. First, is that should the data be stolen you will have given the email, address and password of the users away. This would be a serious breach of data. So we need to hash the password. To do this you can use different methods but the one we will look at here is werkzeug. Import the library:

```
from werkzeug.security import generate_password_hash, check_password_hash
```

The `generate_password_hash(password)` function can be used to take a password and return a string of characters. The password “helloWorld” returns the following hashed value:

```
“pbkdf2:sha256:150000$y9Vk8pT4$e1365d5f8838c9173e80fd63c3f85483ccf052a8774735da4caaee03b0a1b6cb”
```

Step 19: When a user signs up, instead of storing the raw password in your user table you can store the hashed version instead.

Step 20: In addition, you can also check the hash version of a password against the value stored in the database when the user logs in bu using `check_password_hash(pass_hash, password)`. Amend the password validation code of the `LoginForm` (*forms.py*) that it uses this function.