Monday: Flask Login

Flask-Login

The Flask-Login is an extension that helps us manage the user authentication system

```
(virtual)$ pip install flask-login
```

We install the extension and we can Initialize it inside our `create_app` function.

```
#....
from flask_login import LoginManager

login_manager = LoginManager()
login_manager.session_protection = 'strong'
login_manager.login_view = 'auth.login'

def create_app(config_name):
    #....
    #Initializing Flask Extensions
    bootstrap.init_app(app)
    db.init_app(app)
    login_manager.init_app(app)
#....
```

We import the `Login Manager` class from the flask-login extension. We then create an instance of the class.

`login_manager.session_protection` attribute provides different security levels and by setting it to `strong` will monitor the changes in a user's request header and log the user out.

We prefix the login endpoint with the blueprint name because it is located inside a blueprint.

Authenticated routes

Some routes like submitting a new review should only be accessible to users who are authenticated and need to be secured.

app/main/views.py

```
#....
from flask_login import login_required
#....
```

flask_login has a decorator `login_required` that will intercept a request and check if the user is authenticated and if not the user will be redirected to the login page.

app/main/views.py

```
#....
@main.route('/movie/review/new/<int:id>', methods = ['GET','POST'])
@login_required
def new_review(id):
#....
```

We place the decorator after our new review route decorator.

Configuring Models

We need to configure our User model to work with the extension. For this to happen we need our model to implement four methods:

1. is_authenticated() - Returns a boolean if a User is authenticated or not.
2. is_active()- Checks if a user is allowed to authenticate.
3. is_anonymous()- Returns a boolean if a user is anonymous.
4. get_id()- Returns a unique identifier for User.

Flask-login provides a class that has the basic implementations of these methods.

app/models.py

```
from flask_login import UserMixin
```

We import the `UserMixin` class that we will pass into our `User` model. This helps us not to have to implement the methods for ourselves.

app/models.py

```
class User(UserMixin,db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer,primary_key = True)
    username = db.Column(db.String(255),index = True)
    email = db.Column(db.String(255),unique = True,index = True)
    role_id = db.Column(db.Integer,db.ForeignKey('roles.id'))
    password_hash = db.Column(db.String(255))
```

We also add an `email` column to our module, this will help our users to have a better access to their accounts. Remember to create a new Migration after you change the model.

Flask-login also requires a callback function that retrieves a user when a unique identifier is passed.

app/models.py

```
#....
from . import login_manager

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

#....
```

Flask-login has a decorator `@login_manage.user_loader` that modifies the `load_user` function by passing in a `user_id` to the function that queries the database and gets a User with that ID.

Registering New Users

We can now register new Users to our application. We first need to create the registration form. We create a new file inside our `auth` package `forms.py`. We then need to import it to our `auth/___init___.py` file.

`auth/__init__.py`

```
from . import views,forms
```

We can now create our registration form

auth /forms.py

```
from flask_wtf import FlaskForm
from wtforms import StringField,PasswordField,SubmitField
from wtforms.validators import Required,Email,EqualTo
from ..models import User
```

We first need to import a few items. First, we import the `FlaskForm` class from the`flask_wtf` extension. Secondly, we also import some input fields to facilitate user input. We then import some Validators; `Email` validator makes sure that the input follows a proper email address structure and the `EqualTo` helps us in comparing the two password inputs.

auth /forms.py

```
class RegistrationForm(FlaskForm):
    email = StringField('Your Email Address',validators=[Required(),Email()])
    username = StringField('Enter your username',validators = [Required()])
    password = PasswordField('Password',validators = [Required(), EqualTo('password_confirm',
message = 'Passwords must match')])
    password_confirm = PasswordField('Confirm Passwords',validators = [Required()])
    submit = SubmitField('Sign Up')
```

We then create a `RegistrationForm` form class. We create four input fields. `email`for the users email address, and pass in the `Required` and `Email` validators. `username` for the username field, `password` and `password_confirm` fields. We pass in the `EqualTo` validator to the password to make sure both passwords are the same before the form is submitted.

Custom Validators

We can also add our own custom validators.

auth /forms.py

```
from wtforms import ValidationError
class RegistrationForm(FlaskForm):
    # .......
    def validate_email(self,data_field):
            if User.query.filter_by(email =data_field.data).first():
                raise ValidationError('There is an account with that email')

    def validate_username(self,data_field):
        if User.query.filter_by(username = data_field.data).first():
            raise ValidationError('That username is taken')
```

We import `ValidationError` from wtforms and then create two methods.

`validate_email` takes in the data field and checks our database to confirm there is no user registered with that email address. If a user with the same email address is found, a `ValidationError` is raised and the error message passed in is displayed. As a result, the form is not submitted.

`validate_username` checks to see if the username is unique and raises a `ValidationError` if another user with a similar username is found.

When a method inside a form class begins with `validate_` it is considered as a validator and is run with the other validators for that input field.

Registration route

auth/views.py

```
# ....
from flask import render_template,redirect,url_for
from ..models import User
from .forms import RegistrationForm
from .. import db

# ....
@auth.route('/register',methods = ["GET","POST"])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(email = form.email.data, username = form.username.data,password = form.pa
ssword.data)
        db.session.add(user)
        db.session.commit()
        return redirect(url_for('auth.login'))
        title = "New Account"
    return render_template('auth/register.html',registration_form = form)
```

We create the form instance and pass it into our template. When the form is submitted we create a new user from the `User` model and pass in the email,username and password. We add the new user to the session and commit the session to add the user to our database.

Our user can then log in to the application and we redirect them to the login route.

We can render the registration form with the `wtf.quick_form()` function.

app/templates/auth/register.html

```
{% extends 'base.html'%}
{% import 'bootstrap/wtf.html' as wtf%}

{% block content %}
<div class="container">
<div class="row">
<div class="col-md-2"> </div>
<div class="col-md-8">

{{wtf.quick_form(registration_form)}}

</div>
<div class="col-md-2"></div>
</div>
</div>
```

```
{% endblock%}
```

Login Users

We need to create a Login form that will give users access to some of the features of our application.

app/auth/forms.py

```
#....
from wtforms import StringField,PasswordField,BooleanField,SubmitField
.....
```

We add a `BooleanField` input field that will render a checkbox in our form.

app/auth/forms.py

```
class LoginForm(FlaskForm):
    email = StringField('Your Email Address',validators=[Required(),Email()])
    password = PasswordField('Password',validators =[Required()])
    remember = BooleanField('Remember me')
    submit = SubmitField('Sign In')
```

We then create 3 input fields for the users email,password and a boolean to confirm wheter the user wants to be logged out after the session.

app/auth/views.py

```
#....
from flask import render_template,redirect,url_for, flash,request
from flask_login import login_user
from ..models import User
from .forms import LoginForm,RegistrationForm

#....
@auth.route('/login',methods=['GET','POST'])
def login():
    login_form = LoginForm()
    if login_form.validate_on_submit():
        user = User.query.filter_by(email = login_form.email.data).first()
        if user is not None and user.verify_password(login_form.password.data):
            login_user(user,login_form.remember.data)
            return redirect(request.args.get('next') or url_for('main.index'))

        flash('Invalid username or Password')

    title = "watchlist login"
    return render_template('auth/login.html',login_form = login_form,title=title)
#....
```

First, we import flash and request functions from the flask module. The flash function helps us display error messages to the user.

We create an instance of the `LoginForm` and pass it into the `login.html` template. Then we check if the form is validated where we search for a user from our database with the email we receive from the form.

We then use the `verify_password` method to confirm that the password entered matches with the password hash stored in the database. If they match, we use the `login_user` function provided by the `flask_login` extension to record the user as logged for that current session. It takes user object and the remember form data. If True it sets a long time cookie in your browser.

We can now render the form in our `login.html` template using the `wtf.quick_forms()` function.

To confirm login state we can update our navigation bar.

app/templates/navbar.html

```
#....
                <ul class="nav navbar-nav navbar-right">
                  {% if current_user.is_authenticated %}
                  <li style="color:white;padding:15px;">{{current_user.username}}</li>
                  <li><a href="{{url_for('auth.logout')}}">Sign out</a></li>
                  {% else %}
                  <li><a href="{{url_for('auth.login')}}">Sign in</a></li>
                  {%endif%}
                </ul>
#....
```

Current user is a variable that is defined by flask-login and is made available to all the view functions and templates.

Now that we have all that taken care of, let's update our login.html file to have a form and a link to the sign up page.

login.html

```
{% extends 'base.html'%}
{% import 'bootstrap/wtf.html' as wtf%}

{% block content %}
<div class="container">
    <div class="row">
      <div class= "col-md-2"> </div>
      <div class="col-md-8">
      <!--Flash message -->
      {% for message in get_flashed_messages() %}
        <div class="alert alert-danger">
        <button type="button" class="close" data-dismiss="alert">&times;</button>
        {{ message }}
        </div>
      {% endfor %}
      <!-- Login form -->
      {{wtf.quick_form(login_form)}}
      <p> Don't have an account? <a href="{{url_for('auth.register')}}">Sign up here</a></p>
     </div>
     <div clas="col-md-2"> </div>
    </div>
</div>

{% endblock%}
```

Log out

app/auth/views.py

```
from flask_login import login_user,logout_user,login_required
#....
```

```
@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect(url_for("main.index"))

#....
```

We create an authenticated route `logout` that calls the flask_login's `logout_user` function. This will logout the user from our application. We then redirect the user to the `index` page.

You can find the application at this point here **https://github.com/mbuthiya/watchlist/tree/25-creating-registration-form (https://github.com/mbuthiya/watchlist/tree/25-creating-registration-form)**