

## Flask 3: User Accounts with Flask Forms

### Lab Objectives

Objectives of this lab are to continue developing our **Blogging website**, by:

- implementing **user accounts** functionality deploying **Flask forms**;
- and adding **validation** to user authentication functionality.

### PRELIMINARIES

- As before, this exercise is not assessed, but you should complete all tasks.
- **NB:** It is advisable to do all the work with your virtual environment **activated**.
- Review the **PRELIMINARIES** section in Flask 1 instructions - these are relevant to this lab too.

---


#### Useful resources

---

Snapshots demonstrating code at various points of lab tasks completion: <https://git.cardiff.ac.uk/scmne/flask-labs> 

Flask Website: <https://flask.palletsprojects.com/en/2.2.x/> 

Flask-WTF home page: <https://flask-wtf.readthedocs.io/en/1.0.x/> 

Flask-WTF Quickstart: <https://flask-wtf.readthedocs.io/en/1.0.x/quickstart/> 

Also see the 'Useful Resources' section in Flask 1 lab.

---

## USER ACCOUNTS

We want our visitors to be able to create their user accounts, so that they can register, log in and log out. This can also be useful for implementing further functionality later on, e.g. admin login (see the 'Optional Flask Lab' worksheet) .

Implementation of the user accounts is accomplished using:

- **Flask-WTF**<sup>(1)</sup>, which is an integration of Flask and **WTForms**<sup>(2)</sup> - for creating forms.
- **Flask-Login**<sup>(3)</sup>, which will be responsible for handling user authentication.

### Initial Setup and Modifications

1. Make sure you have the virtual environment active.
2. Check if you have **Flask-WTF** package installed, and if not, install it using **pip** in your **venv**.<sup>(4)</sup>

## USER REGISTRATION

To register a new user, we need to create a **user registration** form. It will have the following fields: **username**, **confirm\_username**, **password**, and **submit** button.

3. Create a new file **forms.py** in the **blog** dir.
  - (a) Start with importing **FlaskForm** from **flask\_wtf**:

```
from flask_wtf import FlaskForm
```

- (b) Use **class RegistrationForm(FlaskForm)** to declare the first field **username**, declaring it as a String and checking the input is 'true' value/non-empty.

```
...  
class RegistrationForm(FlaskForm):  
    username = StringField('Username', validators=[DataRequired()])
```

This statement requires two imports, such as: **StringField** from **wtforms**, and **DataRequired** from **wtforms.validators**, i.e.:

```
from wtforms import StringField  
from wtforms.validators import DataRequired
```

- (c) Using appropriate data types and validators, add the definitions for: **password** field and **submit** button. The labels for these fields should be '*Password*' and '*Register*', respectively. Your definition for the **password** field should also check for 'true' value input. Consult WTForms documentation on fields<sup>(5)</sup> and validators<sup>(6)</sup> or lecture

---

(1) <https://flask-wtf.readthedocs.io/>

(2) <https://wtforms.readthedocs.io/>

(3) <https://flask-login.readthedocs.io/>

(4) If you are getting an error message when you use **pip** to install a python package, you might need to use **--user** option, i.e. **pip install --user <PACKAGE>**.

notes to help you complete this task.

4. The next step is to add the form to the registration page - `register.html`.

(a) Create `register.html` in `blog/templates` dir.

(b) In `{% block content %}` section, specify that we want to add the form and its field `username` (7):

```
...
<form method="POST" action="">
    {{ form.csrf_token }}
    {{ form.username.label }} {{ form.username }}
    ...

    <input type="submit" value="Register">
</form>
...
```

(c) Using the same principle, add `password` form fields we specified in our `user registration` form.

5. To process the form, we need to modify `routes.py` to tell the server how to handle the form:

(a) Import `RegistrationForm` class from `forms.py`:

```
...
from blog.forms import RegistrationForm
...
```

(b) Add the `@app.route` decorator for `register`:


```
...
@app.route("/register", methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if request.method == 'POST':
        user = User(username=form.username.data, password=form.password.data)
        db.session.add(user)
        db.session.commit()
    return render_template('register.html', title='Register', form=form)
...
```

**[NB]** Make sure to import `request` from `flask`.

---

(5) <https://wtforms.readthedocs.io/en/3.0.x/fields/> 

(6) <https://wtforms.readthedocs.io/en/3.0.x/validators/> 

(7) If you are curious about what "`form.csrf_token`" is about, see here: <https://wtforms.readthedocs.io/en/3.0.x/csrf/> 

6. On successful registration, we want to redirect the newly registered user to a *'Thanks for registering with us!'* page. <sup>(8)</sup>

(a) Create an appropriate `registered.html` in `templates` dir.

(b) In `routes.py`, create `registered` decorator:

```
@app.route("/registered")
def registered():
    return render_template('registered.html', title='Thanks!')
```

and then add redirect instruction to `register` decorator:

```
...
    return redirect(url_for('registered'))
...
```

**[NB]** The above requires import of `redirect` from `flask`.

7. Test the registration process works as intended, by going to `http://127.0.0.1:5000/register` and create a few 'users'.

*Note:* For the moment, we are assuming the user only provides valid input. Handling user input's validity and errors is dealt with in *'FORM VALIDATION'* section of this document.

8. Check the registered users' details are now listed in your db, in `user` table, e.g.:

```
sqlite> SELECT * FROM user;
```

id	username	password
1	jane_doe	password123
2	john_doe	qwerty123

---

(8) This will also play the role of a test to check the registration works, albeit a simple one!

## USER LOGIN

**User Login** functionality is implemented, using similar principles to those followed when implementing **User Registration**, i.e. you need to create `LoginForm` class in `forms.py`, create `login.html` template, and add logic of how to handle login to `routes.py`. We also need to update the `User` model in `models.py`

We will be using `Flask-Login` package<sup>(9)</sup> to take care of user session management in our app, which you need to install (if you haven't done yet.)

9. In `__init__.py` in `blog` dir:

- (a) Add an import for `LoginManager` and initialise its object:

```
...
from flask_login import LoginManager
...
login_manager = LoginManager()
login_manager.init_app(app)
...
```

- (b) Provide a `user_loader` callback, which will be used to load the user object used on its identifier. Place the following code in `models.py`

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

And import `login_manager`:

```
...
from blog import db, login_manager
...
```

10. Modify the `User` definition to enable us to use various methods and properties implemented in `UserMixin` class, e.g. to check a user `is_active` or `is_authenticated`<sup>(10)</sup>

```
...
from flask_login import UserMixin
...

class User(UserMixin, db.Model):
    ...
```

11. Update `forms.py`, by creating `LoginForm`:

```
class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    submit = SubmitField('Login')
```

---

(9) <https://flask-login.readthedocs.io/en/latest/> 

(10) <https://flask-login.readthedocs.io/en/latest/#your-user-class>

12. Create a new `login.html`, and add the form to display the user's `username` and `password` fields (similar to what we did for Task 4).

13. Modify `routes.py`:

- (a) Add `@app.route("/login")`:

```
@app.route("/login",methods=['GET','POST'])
def login():
    form = LoginForm()
    if request.method=='POST':
        user = User.query.filter_by(username=form.username.data).first()
        login_user(user)
        return redirect(url_for('home'))
    return render_template('login.html',title='Login',form=form)
```

- (b) Add the necessary *imports*, such as `LoginForm` and `login_user`:

```
...
from blog.forms import RegistrationForm, LoginForm
from flask_login import login_user
...
```

14. Test the login process works as intended, by going to `http://127.0.0.1:5000/login` and check you can log in as one of the users you added to the database in Task 7. On successful login the website will return to `home` page.

**[NB]** As before, for the moment, we are assuming the user only provides valid input. Handling user input validity and errors is covered in the next section.

## User Logout

15. You might be pleased to know that all your hard work is done by now - 'user logout' functionality is, perhaps, the easiest of all to implement! We don't need to have a special form for this, just modification of `routes.py`, to which we need to add the `@app.route("/logout")` decorator, and Flask will handle the logout without us needing to implement it 'from the ground up':

```
...
@app.route("/logout")
def logout():
    logout_user()
...
```

**[NB]** Make sure to add the necessary import(s) from `flask_login`.

16. Modify the code to redirect the user to `home` page after they have been successfully logged out.
17. Finally for this part of the exercise, modify the website's navigation to include links to `register`, `login` and `logout` on each page.

## FORM VALIDATION

So far, we have already provided some constraints in our models. However, validation we implemented was minimal. We assumed that the users would always provide input that is valid. If they don't, the system will behave unexpectedly and possibly crash. For example, in our system, we use `username` for registration and login. If a user tries to register with a username that already exists or log in without being registered, the system will most likely throw a *'database integrity error'*. To avoid issues like this, we need to validate the users' input and provide them with hints and help, e.g. we want to inform the user if the username they provide during the registration is already taken. In this section, we will define logic and functions for user input validation and bind these to appropriate URLs, as well as carry out other improvements to enhance the system's functionality and security.

**[NB]** In this lab, we are using Flask to validate the forms. Alternatively, you might want to look into using JavaScript or other technologies and tools to achieve this. This, however, is out of scope for these labs and presents an opportunity for independent learning.

## DB Update

18. We store the user's **login credentials** in our db. However, the passwords are currently stored in a plain text form and are visible to the 'naked eye'. To improve the password security we will use password hashing function. To do this we need to modify an existing table `User` for the db, by: (a) modifying `models.py`, and (b) writing the changes to the db:

- (a) Firstly, we modify `User` class attributes (i.e. columns stored in the db) by replacing plain text `password` with a hashed one:

```
...
class User(UserMixin,db.Model):
    ...
    ...
    hashed_password = db.Column(db.String(128))
    ...
    def __repr__(self):
        return f"User('{self.username}', '{self.email}')"

    @property
    def password(self):
        raise AttributeError('Password is not readable.')

    @password.setter
    def password(self,password):
        self.password_hash=generate_password_hash(password)

    def verify_password(self,password):
        return check_password_hash(self.password_hash,password)
```

The above code requires a number of *imports* from `werkzeug.security`, namely: `generate_password_hash` and `check_password_hash`.

(b) Secondly, we need to redo `user` table in the db:

- i. Back up the db first - in case you want or need to revert to it.
- ii. Drop `user` table, e.g. in `sqlite3` CLI, the command is:

```
sqlite> DROP TABLE user;
```

**[NB]** If you have populated `post` table it might be a good idea to drop it as well to avoid problems with the db and website.

- iii. In the python shell <sup>(11)</sup>, update the db using the following commands:

```
> python
>>> from blog import db
>>> db.create_all()
```

- iv. Confirm that you now have the `User` table updated. The command to show the schema of a particular table in `sqlite3` CLI is:

```
-- to describe a specific table: .schema TABLE
sqlite> .schema user
```

This will display SQL's `CREATE TABLE` statement.

To view the table schema in a more user friendly output, use:

```
sqlite> .header ON
sqlite> .mode columns
sqlite> PRAGMA table_info(user);
```

The table `user` should now look like:

cid	name	type	notnull	dflt_value	pk
0	id	INTEGER	1		1
1	username	VARCHAR(15)	1		0
2	hashed_password	VARCHAR(128)	0		0

---

(11) Similar to what we did in the previous lab - see `DATABASE (DB)` section of *Flask 2* lab. Alternatively, you might want to use any other suitable method, e.g. a GUI.



19. Let's check `username` already exists. In class `RegistrationForm` (`forms.py`), we would specify this rule as:

```
...
def validate_username(self, username):
    user = User.query.filter_by(username=username.data).first()
    if user is not None:
        raise ValidationError('Username already exist. Please choose a different
        ↪ one.')
```

**[NB]** The above code requires an import of `ValidationError` from `wtforms.validators` (which we have already imported earlier), and an import of `User` from `blogs.models`.

20. We could also specify certain rules for the usernames. Suppose we want to reinforce the following rule: a username must contain minimum of 6 and maximum of 8 of low case letter characters, i.e. digits or non-alphanumeric characters are not allowed, e.g. `abcdef` will be valid, but not `abc`, `Abc`, `abc1f!` or `abcdefghi`.<sup>(12)</sup>

We can use a regular expression (regex) for this. Update `username` in `forms.py`, as follows:

```
...
username = StringField('Username', validators=[DataRequired(),
    ↪ Regexp('[a-z]{6,8}$', message='Your username should be between 6 and 8
    ↪ characters long, and can only contain lowercase letters.')]
...

```

**[NB]** Don't forget to import `Regexp` from `wtforms.validators`.

If you want to learn more about regex:

- [https://www.w3schools.com/python/python\\_regex.asp](https://www.w3schools.com/python/python_regex.asp)  has a number of examples which you can try out.

21. Lets' also ask the user to confirm their username to make sure they have not made a mistake. We can use `EqualTo` validator from `wtforms.validators`, which we can chain on to `Regexp` validator:

```
...
username = StringField('Username', validators=[DataRequired(),
    ↪ Regexp(...), EqualTo('confirm_username', message='Usernames do not match. Try
    ↪ again')])
...

```

We will then need to add this field to `forms.py` and `register.html`.

---

(12) This, of course, is a very simple requirement for the username. If we want to make the rule more complicated, we would need to use a more complicated regex. For instance, if we want to make sure that a user's password must be between 6 and 8 characters long AND contains at least one numeric digit, the regex for this would be: `^(?=.*\d){6,8}$`.

## Routing

22. After we specified the logic for validation, we need to add checking the form is valid when it is submitted, so in `routes.py` instead of using

```
if request.method == 'POST':
```

we need to to use:

```
if form.validate_on_submit():
```

in the appropriate `@app.route(..)` decorators for `registration` and `login`.

## Error Messages

Any good system should provide its user with feedback. We have already encountered that we can specify a message to the user during form validation (Task 20). We can also use a messaging system provided by Flask, called '*flashing system*'.

**[NB]** This functionality requires an import of `flash` from `flask` in `routes.py`.

23. The following are examples of flash messages, which you can add to appropriate `@app.route` decorators in `routes.py`:

```
...
flash('Registration successful!')
...
flash('Invalid email address or password.')
...
flash('You\'ve successfully logged in.')
...
flash('Logout successful. Bye!')
```


24. To enable the '*flashing system*', we need to add code to the templates to instruct the server to display the messages:

(a) **site-wide**, by adding the following to `layout.html`:

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

(b) and on a *specific page*, e.g. in `register.html`:

```
{% for error in form.username.errors %}
    <span style="color: red;">[{{ error }}]</span>
{% endfor %}
```

**[NB]** More information on *Message Flashing* in Flask can be found at: <https://flask.palletsprojects.com/en/2.0.x/patterns/flashing/> .

## FURTHER ENHANCEMENT

### Validation Enhancement

25. Enhance validation and error messaging for each of the user account functionality – *registration*, *login* and *logout* – by implementing additional functionality, which you think is appropriate and necessary, e.g. customisation of the Flask's default '*empty field*' error message, displaying useful help messages as the user steps through the input field when completing the registration form. These are just suggestions, there are many other possible useful and appropriate extensions to user input validation, which would enhance the users' overall experience with the system.

### Usability and Navigation Enhancements

26. Modify `layout.html` and other pages to enhance the website usability by improving navigation, such as providing your website visitors with the links to `register`, `login` and `logout`. You can also make the users experience more compelling by displaying personalised messages and navigation menu(s), modifying the current redirection logic, etc.
-